# Chapter 2
# ML, a Functional Programming Language

(Version of 24 September 2004)

# 2.1. Expressions

## *Interacting with ML*

```
-   32 + 15 ;
  val it = 47 : int

-   3.12 * 4.3 ;
  val it = 13.416 : real

-   not true ;
  val it = false : bool

-   "The Good, the Bad," ^ " and the Ugly" ;
  val it = "The Good, the Bad, and the Ugly" : string

-   ( size("Esra") +
=      size("Pierre") ) div 2 ;
  val it = 5 : int
```

- ML has an *interpreter*
- ML is a *typed* language

## *Basic types*

- **unit**: only one possible value: ()
- **int**: integers
- **real**: real numbers
- **bool**: truth values (or: Booleans) **true** and **false**
- **char**: characters
- **string**: character sequences

## *Operators*

- We use *operator* and *function* as synonyms
- We use *argument*, *parameter*, and *operand* as synonyms

## *Operator types*

```
-   2 + 3.5 ;
  ! 2 + 3.5 ;
  !     ^^^
  ! Type clash: expression of type real
  !                    cannot have type int
```

The operators on the basic types are thus *typed*:
no mixing, no implicit conversions!
For convenience, the arithmetic operators are *overloaded*:
the same symbol is used for different operations,
but they have different realisations; for instance:
$+ : \text{int} \times \text{int} \rightarrow \text{int}$
$+ : \text{real} \times \text{real} \rightarrow \text{real}$

# Integers

*Syntax*

- As usual, except the unary operator $-$ is represented by ˜
- Example: ˜123

*Basic operators on the integers*

| *op* | : | *type* | *form* | *precedence* |
|---|---|---|---|---|
| $+$ | : | $\text{int} \times \text{int} \to \text{int}$ | infix | 6 |
| $-$ | : | $\text{int} \times \text{int} \to \text{int}$ | infix | 6 |
| $*$ | : | $\text{int} \times \text{int} \to \text{int}$ | infix | 7 |
| **div** | : | $\text{int} \times \text{int} \to \text{int}$ | infix | 7 |
| **mod** | : | $\text{int} \times \text{int} \to \text{int}$ | infix | 7 |
| $=$ | : | $\text{int} \times \text{int} \to \text{bool}$ * | infix | 4 |
| $<>$ | : | $\text{int} \times \text{int} \to \text{bool}$ * | infix | 4 |
| $<$ | : | $\text{int} \times \text{int} \to \text{bool}$ | infix | 4 |
| $<=$ | : | $\text{int} \times \text{int} \to \text{bool}$ | infix | 4 |
| $>$ | : | $\text{int} \times \text{int} \to \text{bool}$ | infix | 4 |
| $>=$ | : | $\text{int} \times \text{int} \to \text{bool}$ | infix | 4 |
| ˜ | : | $\text{int} \to \text{int}$ | prefix | |
| **abs** | : | $\text{int} \to \text{int}$ | prefix | |

(* the exact type will be defined later)

- The infix operators associate to the left
- Their operands are always *all* evaluated

# Real numbers

## *Syntax*

- As usual, except the unary operator − is represented by ˜
- Examples: **234.2** , **˜12.34** , **˜34E2** , **4.57E˜3**

## *Basic operators on the reals*

| *op* | : | *type* | *form* | *precedence* |
|---|---|---|---|---|
| $+$ | : | real × real → real | infix | 6 |
| $-$ | : | real × real → real | infix | 6 |
| $*$ | : | real × real → real | infix | 7 |
| $/$ | : | real × real → real | infix | 7 |
| $=$ | : | real × real → bool * | infix | 4 |
| $<>$ | : | real × real → bool * | infix | 4 |
| $<$ | : | real × real → bool | infix | 4 |
| $<=$ | : | real × real → bool | infix | 4 |
| $>$ | : | real × real → bool | infix | 4 |
| $>=$ | : | real × real → bool | infix | 4 |
| ˜ | : | real → real | prefix | |
| **abs** | : | real → real | prefix | |
| **Math.sqrt** | : | real → real | prefix | |
| **Math.ln** | : | real → real | prefix | |

(* the exact type will be defined later)

- The infix operators associate to the left
- Their operands are always *all* evaluated

# Characters and strings

## *Syntax*

- A *character* value is written as the symbol `#` immediately followed by the character enclosed in double-quotes `"`
- A *string* is a character sequence enclosed in double-quotes `"`
- Control characters can be included:
  end-of-line: `\n` double-quote: `\"` backslash: `\\`

## *Basic operators on the characters and strings*

Let 'strchar × strchar' be 'char × char' or 'string × string'

| *op* | : | *type* | *form* | *precedence* |
|---|---|---|---|---|
| = | : | strchar × strchar → bool * | infix | 4 |
| <> | : | strchar × strchar → bool * | infix | 4 |
| < | : | strchar × strchar → bool | infix | 4 |
| <= | : | strchar × strchar → bool | infix | 4 |
| > | : | strchar × strchar → bool | infix | 4 |
| >= | : | strchar × strchar → bool | infix | 4 |
| ^ | : | string × string → string | infix | 6 |
| **size** | : | string → int | prefix | |

(* the exact type will be defined later)

Use of the *lexicographic order*, according to the ASCII code

- The infix operators associate to the left
- Their operands are always *all* evaluated

# Booleans

## *Syntax*

- Truth values **true** and **false**
- Attention: **True** is *not* a value of type bool:
  ML distinguishes uppercase and lowercase characters!

## *Basic operators on the Booleans*

| *op* | : | *type* | *form* | *precedence* |
|---|---|---|---|---|
| **andalso** | : | bool × bool → bool | infix | 3 |
| **orelse** | : | bool × bool → bool | infix | 2 |
| **not** | : | bool → bool | prefix | |
| = | : | bool × bool → bool * | infix | 4 |
| <> | : | bool × bool → bool * | infix | 4 |

(* the exact type will be defined later)

## *Truth table*

| A | B | A **andalso** B | A **orelse** B |
|---|---|---|---|
| **true** | **true** | **true** | **true** |
| **true** | **false** | **false** | **true** |
| **false** | **true** | **false** | **true** |
| **false** | **false** | **false** | **false** |

- The infix operators associate to the left
- The second operand of **andalso** & **orelse** is *not* always evaluated: *lazy* logical *and* & *or*

Example:

```
-   ( 34 < 649 )  orelse  (  Math.ln(12.4) * 3.4 > 12.0 ) ;
  val it = true : bool
```

The second operand, namely ( **Math.ln**(12.4) ∗ 3.4 > 12.0 ), is *not* evaluated because the first operand evaluates to **true**

Another example:

```
-   ( 34 < 649 )  orelse  ( 0.0 / 0.0 > 999.9 ) ;
  val it = true : bool
```

The second operand ( 0.0 / 0.0 > 999.9 ) is *not* evaluated, even though by itself it would lead to an error:

```
-   ( 0.0 / 0.0 > 999.9 ) ;
  ! Uncaught exception: Div
```

# Type conversions

| *op* | : | *type* |
|---|---|---|
| **real** | : | int → real |
| **ceil** | : | real → int |
| **floor** | : | real → int |
| **round** | : | real → int |
| **trunc** | : | real → int |

```
-   real(2) + 3.5 ;
  val it = 5.5 : real
-   ceil(23.65) ;
  val it = 24 : int
-   ceil(~23.65) ;
  val it = ~23 : int
-   floor(23.65) ;
  val it = 23 : int
-   floor(~23.65) ;
  val it = ~24 : int
-   round(23.65) ;
  val it = 24 : int
-   round(23.5) ;
  val it = 24 : int
-   round(22.5) ;
  val it = 22 : int
```

```
-   trunc(23.65) ;
    val it = 23 : int
-   trunc(~23.65) ;
    val it = ~23 : int
```

| *op* | : | *type* |
|---|---|---|
| **chr** | : | int $\rightarrow$ char |
| **ord** | : | char $\rightarrow$ int |
| **str** | : | char $\rightarrow$ string |

```
-   chr(97) ;
    val it = #"a" : char
-   ord(#"a") ;
    val it = 97 : int
-   str(#"a") ;
    val it = "a" : string
```

Conversions are done according to the ASCII code

# Evaluation of expressions

*Reduction*

```
3 + 4 * 2  <  5 * 2
⇝  3 + 8  <  5 * 2
⇝  11  <  5 * 2
⇝  11  <  10
⇝  false
```

- Note the precedence of the operators
- Reduction to a *normal form*
  (a form that cannot be further reduced)
- This normal form is the *result* of the evaluation
- The type of the result is inferred from those of the operators

*Principles*

Reduction (evaluation) of the expression $E_1$ `op` $E_2$

1. Reduction of the expression $E_1$: $E_1 \leadsto \ \ldots \ \leadsto \ N_1$
2. Reduction of the expression $E_2$: $E_2 \leadsto \ \ldots \ \leadsto \ N_2$
   unless `op` is lazy and $N_1$ is such that $E_2$ need not be reduced
3. Application of the operator `op` to $N_1$ and $N_2$

Evaluation from left to right: first $E_1$ then $E_2$ (if necessary)

# Conditional expressions

```
-   if 3 >= 0 then 4.1 + 2.3 else 2.1 / 0.0 ;
  val it = 6.4 : real
```

*Reduction*

**if** 3 >= 0 **then** 4.1 + 2.3 **else** 2.1 / 0.0
$\rightsquigarrow$ **if** **true** **then** 4.1 + 2.3 **else** 2.1 / 0.0
$\rightsquigarrow$ 4.1 + 2.3
$\rightsquigarrow$ 6.4

*Principles*

In the expression **if** $BExpr$ **then** $Expr_1$ **else** $Expr_2$

- $BExpr$ must be a Boolean expression
- $Expr_1$ and $Expr_2$ must be expressions of the same type

Reduction:

- $Expr_1$ is only evaluated if $BExpr$ evaluates to **true**
- $Expr_2$ is only evaluated if $BExpr$ evaluates to **false**

*Remarks*

- Note that **if** ... **then** ... **else** ... is an *expression*, but *not* a control structure
- There is no **if** ... **then** ... in functional languages: such an expression would be meaningless when its test (the Boolean expression) evaluates to **false**

# 2.2. Value declarations

*Examples*

```
-  val pi = 3.14159 ;
   val pi = 3.14159 : real

-  val twoPi = 2.0 * pi ;
   val twoPi = 6.28318 : real

-  twoPi * 5.3 ;
   val it = 33.300854 : real

-  it / 2.0 ;
   val it = 16.650427 : real

-  val &@!+<% = "bizarre, no?!" ;
   val &@!+<% = "bizarre, no?!" : string
```

*Identifiers*

- Alphanumeric identifiers
- Symbolic identifiers
  made from `+ - / * < > = ! @ # $ % ^ & ` ` ` `~ \ | ? :`
- Do not mix alphanumeric and symbolic characters
- The identifier `it` always has the result of the
  last unidentified expression evaluated by the interpreter
- Attention: `3 +~ 2` is *different* from `3 + ~2`
  One must separate the symbols `+` and `~` with a space,
  otherwise they form a new symbolic identifier

## *Bindings and environments*

- The execution of a declaration, say **val** x = *expr*,
  creates a *binding*:
  the identifier x is *bound* to the value of the expression *expr*
- A collection of bindings is called an *environment*
- The identifier it is always bound to the result of the last unidentified expression evaluated by the interpreter

## *Identifiers vs. variables*

```
-   val sum = 24 ;
  val sum = 24 : int
-   val sum = 3.51 ;
  val sum = 3.51 : real
```

- Association of a value to an identifier
- In ML, there are only "variables" in the mathematical sense
- No assignment,
  no variables (in the imperative-programming sense),
  no "modification" of variables

## *Evaluation order*

```
-   val a = 1 ;
  val a = 1 : int
-   val b = 2 ;
  val b = 2 : int
-   val a = 1  val b = 2 ;
  val a = 1 : int
  val b = 2 : int
-   val a = a+b  val b = a+b ;
  val a = 3 : int
  val b = 5 : int
```

- Evaluation and declaration from left to right

```
-   val a = 1  val b = 2 ;
  val a = 1 : int
  val b = 2 : int
-   val a = a+b  and b = a+b ;
  val a = 3 : int
  val b = 3 : int
```

1. Simultaneous evaluation of the right-hand sides of the declarations
2. Declaration of the identifiers

# 2.3. Function declarations

*Example*

```
-   (* Absolute value of x *)
=   fun abs( x : int ) : int =
=   if x >= 0 then x else ~ x ;
  val abs = fn : int -> int

-   abs(~3) ;
  val it = 3 : int
```

- The argument of a function is *typed*
- The result of a function is also typed
- int → int is the type of functions from integers to integers
- A truth-valued (or: Boolean) function is called a *predicate*

*Evaluation: reduction*

```
abs(3-6)
⇝ abs(~3)
⇝ if ~3 >= 0 then ~3 else ~(~3)
⇝ if false then ~3 else ~(~3)
⇝ ~(~3)
⇝ 3
```

The argument is always evaluated *before* applying the function: *value passing*

# Usage of functions

*Example*

```
-  fun signSquare( x : int ) : int = abs(x) * x ;
   val signSquare = fn : int -> int

-  signSquare(~3) ;
   val it = ~9 : int
```

- The used function abs must have been declared beforehand
- Possibility of simultaneous declarations:

```
-  fun signSquare( x : int ) : int = abs(x) * x
=  and abs( x : int ) : int = if x >= 0 then x else ~ x ;
   val signSquare = fn : int -> int
   val abs = fn : int -> int
```

*Evaluation: reduction*

signSquare(3−6)
$\leadsto$ signSquare(~3)
$\leadsto$ abs(~3) * ~3
$\leadsto$ (**if** ~3 >= 0 **then** ~3 **else** ~(~3)) * ~3
$\leadsto$ (**if** **false** **then** ~3 **else** ~(~3)) * ~3
$\leadsto$ ~(~3) * ~3
$\leadsto$ 3 * ~3
$\leadsto$ ~9

# 2.4. Type inference

In ML, it is often unnecessary to explicitly indicate the type of the argument and result:
their types are *inferred* by the ML interpreter!

*Example*

```
-   fun abs( x ) =
=       if x >= 0 then x else ~ x ;
  val abs = fn : int -> int
```

From `x >= 0` , the ML interpreter infers
that `x` must necessarily be of type `int`
because the type `int` of `0` is recognised from the syntax;
hence the result of `abs` must be of type `int`

If a type cannot be inferred from the context,
then the default is that an
overloaded operator symbol refers to the function on integers

*Example*

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

It is necessary to give enough clues for the type inference:
it is better to give too many clues than not enough!

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

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

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

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

-  fun square( x ) = x : real * x ;
   ! fun square( x ) = x : real * x;
   !                             ^
   ! Unbound type constructor: x
```

The operator ‘ : ’ has a lower precedence than ‘ * ’,
so `x : real * x` is interpreted as `x : ( real * x )`

When using the overloaded operators (+, *, <, . . . ),
it is often necessary to indicate the types of the operands

# 2.5. Anonymous functions

Just like integers and reals, functions are objects!

One can declare and use a function without naming it:

```
-  fun double( x ) = 2 * x ;
   val double = fn : int -> int

-  val double = fn x => 2 * x ;
   val double = fn : int -> int

-  double ;
   val it = fn : int -> int

-  double(3) ;
   val it = 6 : int

-  fn x => 2 * x ;
   val it = fn : int -> int

-  (fn x => 2 * x)(3) ;
   val it = 6 : int
```

The forms **fun** $Name$ $Arg$ = $Def$
and **val** $Name$ = **fn** $Arg$ => $Def$
are equivalent!

## *Usefulness of anonymous functions*

- For higher-order functions (with functional arguments)
- Understanding the reduction of the application of a function

## *Reduction*

double(3) + 4
$\rightsquigarrow$ (**fn** x => 2 $*$ x)(3) + 4
$\rightsquigarrow$ (2 $*$ 3) + 4
$\rightsquigarrow$ 6 + 4
$\rightsquigarrow$ 10

- Function application has precedence 8
- The argument can follow the function name *without* being between parentheses!

## *Principles*

Reduction of $E_1\ E_2$

1. Reduction of the expression $E_1$: $E_1 \rightsquigarrow\ \ldots\ \rightsquigarrow\ N_1$
   $N_1$ must be of the form **fn** $Arg$ => $Def$
2. Reduction of the expression $E_2$: $E_2 \rightsquigarrow\ \ldots\ \rightsquigarrow\ N_2$
3. Application of $N_1$ to $N_2$:
   replacement in $Def$ of all occurrences of $Arg$ by $N_2$
4. Reduction of the result of the application

# 2.6. Specifications

*How to specify an ML function?*

- *Function name* and *argument*
- *Type* of the function: types of the argument and result
- *Pre-condition* on the argument:
  - If the pre-condition does not hold, then the function *may* return *any* result!
  - If the pre-condition does hold, then the function *must* return a result satisfying the post-condition!
- *Post-condition* on the result: its description and meaning
- *Side effects* (if any): printing of the result, …
- *Examples* and *counter-examples* (if useful)

*Example*

**function** sum n
TYPE: int $\rightarrow$ int
PRE: n $\geq$ 0
POST: $\sum_{0 \leq i \leq n} i$

*Beware*

- The post-condition and side effects *should* involve *all* the components of the argument

*Role of well-chosen examples and counter-examples*

In theory:

- They are redundant with the pre/post-conditions

In practice:

- They often provide an intuitive understanding that no assertion or definition could achieve
- They often help eliminate risks of ambiguity in the pre/post-conditions by illustrating delicate issues
- If they contradict the pre/post-conditions, then we know that something is wrong somewhere!

*Example*

**function** floor n
TYPE: real $\longrightarrow$ int
PRE: (none)
POST: the largest integer m such that m $\leq$ n
EXAMPLES: floor(23.65) = 23, floor(˜23.65) = ˜24
COUNTER-EXAMPLE: floor(˜23.65) $\neq$ ˜23

# 2.7. Tuples and records

*Tuples*

- Group $n$ values of possibly different types into *$n$-tuples* by enclosing them in parentheses, say: `(22>5, "abc", 123)`
- Particular cases of $n$-tuples: pairs (or: couples), triples, ...
- Careful: There are *no* 1-tuples in ML!

*Example*

```
-  (2.3, 5) ;
val it = (2.3, 5) : real * int
```

- Operator $*$ here means the Cartesian product of types
- Selector $\#i$ returns the $i^{\text{th}}$ *component* of a tuple
- It is possible to have tuples of tuples
- The value `()` is the only 0-tuple, and it has type **unit**
- The expression ($e$) is equivalent to $e$, hence *not* a 1-tuple! Example: `sum(n)` can also be written as `sum n`

```
-  val bigTuple = ((2.3, 5), "two", (8, true)) ;
val bigTuple = ((2.3, 5), "two", (8, true)) :
        (real * int) * string * (int * bool)

-  #3 bigTuple ;
val it = (8, true) : int * bool

-  #2(#1 bigTuple) + #1(#3 bigTuple) ;
val it = 13 : int
```

## *Records*

- A *record* is a generalised tuple where each component is identified by a *label* rather than by its integer position, and where curly braces are used instead of parentheses
- A record component is also called a *field*

## *Example*

```
-  {course = "FP", year = 2} ;
 val it = {course = "FP", year = 2} :
          {course : string, year : int}
```

- Selector #*label* returns the value of the component identifed by *label*
- It is possible to have records of records
- $n$-tuples are just records with integer labels (when $n \neq 1$)

```
-  #a {a=1, b="xyz"} ;
 val it = 1 : int

-  {a=1, b="xyz"} = {b="xyz", a=1} ;
 val it = true : bool

-  (1, "xyz") = ("xyz", 1) ;
 ! (1, "xyz") = ("xyz", 1);
 !                 ^^^^^
 ! Type clash: expression of type string
 ! cannot have type int

-  {1=1, 2="xyz"} = (1, "xyz") ;
 val it = true : bool
```

# 2.8. Functions with several arguments/results

In ML, a function always has:

- a unique argument
- a unique result

*"Multiple-argument" functions*

```
-   fun max (a,b) = if a > b then a else b ;
  val max = fn : int * int -> int
```

The function max has *one* argument, which is a pair

*"Multiple-result" functions* (divCheck.sml)

```
-   fun divCheck (a,b) =
=        if b = 0 then (0, true, "division by 0")
=        else (a div b, false, "") ;
  val divCheck = fn : int * int -> int * bool * string
-   divCheck (3,0) ;
  val it = (0, true, "division by 0") :
           int * bool * string
```

The function divCheck has *one* result, which is a triple

## *Functions "without arguments or results"*

The basic type **unit** allows us to "simulate" functions that have no arguments or no results

```
-  fun const10 () = 10 ;
   val const10 = fn : unit -> int

-  const10 () ;
   val it = 10 : int

-  const10 ;
   val it = fn : unit -> int

-  fun useless (n:int) = () ;
   val useless = fn : int -> unit

-  useless 23 ;
   val it = () : unit
```

# 2.9. Currying

There is equivalence of the types of the following functions:

$$f : A \times B \rightarrow C$$

$$g : A \rightarrow (B \rightarrow C)$$

H.B. Curry (1958): $f\ (a, b)\ =\ g\ a\ b$

*Currying* = passing from the first form to the second form

Let $a$ be an object of type $A$, and $b$ an object of type $B$

- $f\ (a, b)$ is an object of type $C$
  Application of the function $f$ to the pair $(a, b)$
- $g\ a$ is an object of type $B \rightarrow C$
  $g\ a$ is thus a *function*
  A function is an ML object, just like an integer:
  the result of a function can thus also be a function!
- $(g\ a)\ b$ is an object of type $C$
  Application of the function $g\ a$ to $b$
- Attention: $f\ (a, b)$ is different from $f\ a\ b$

*Principle*

Every function on a Cartesian product can be curried:

$$g : A_1 \times A_2 \times \cdots \times A_n \to C$$

$$\downarrow$$

$$g : A_1 \to (A_2 \to \cdots \to (A_n \to C))$$
$$g : A_1 \to A_2 \to \cdots \to A_n \to C$$

The symbol $\to$ associates to the *right*

*Usefulness of currying*

- The rice tastes better …
- Partial application of a function for getting other functions
- Easier design and usage of higher-order functions (functions with functional arguments)

*Example* (`log.sml`)

**function** log base x
TYPE: int $\longrightarrow$ real $\longrightarrow$ real
PRE: base > 0
POST: $\log_{base}$ x

**fun** log base x = Math.ln x / Math.ln (real base)

```
-  log 2 12.3 ;
   val it = 3.62058641045 : real
-  fun logTwo x = log 2 x ;
   val logTwo = fn : real -> real
-  logTwo 16.0 ;
   val it = 4.0 : real
```

Reduction

log 2 16.0
$\leadsto$ (**fn** base => (**fn** x => Math.ln x / Math.ln (real base))) 2 16.0
$\leadsto$ (**fn** x => Math.ln x / Math.ln (real 2)) 16.0
$\leadsto$ Math.ln 16.0 / Math.ln (real 2)
$\leadsto$ 2.77258872224 / Math.ln (real 2)
$\leadsto$ 2.77258872224 / Math.ln 2.0
$\leadsto$ 2.77258872224 / 0.69314718056
$\leadsto$ 4.0

logTwo 16.0
⇝ (**fn** x => log 2 x) 16.0
⇝ log 2 16.0
⇝ (**fn** base => (**fn** x => Math.ln x / Math.ln (real base))) 2 16.0
⇝ ...

The currying of log is irrelevant here

```
-   log 2 12.3 ;
  val it = 3.62058641045 : real
-   val logTwoBis = log 2 ;
  val logTwoBis = fn : real -> real
-   logTwoBis 16.0 ;
  val it = 4.0 : real
```

log 2
⇝ (**fn** base => (**fn** x => Math.ln x / Math.ln (real base))) 2
⇝ (**fn** x => Math.ln x / Math.ln (real 2))

logTwoBis 16.0
⇝ (**fn** x => Math.ln x / Math.ln (real 2)) 16.0
⇝ Math.ln 16.0 / Math.ln (real 2)
⇝ ...

The currying of log is essential here
Why can logTwoBis not be declared with **fun** rather than **val** ?

# 2.10. Pattern matching and case analysis

*Pattern matching*

```
-  val x = (18, true) ;
   val x = (18, true) : int * bool

-  val (n, b) = (18, true) ;
   val n = 18 : int
   val b = true : bool

-  val (n, _) = (18, true) ;
   val n = 18 : int

-  val (n, true) = x ;
   val n = 18 : int

-  val (n, false) = x ;
   ! Uncaught exception: Bind
```

- The left-hand side of a value declaration is called a *pattern* and must contain (in this case) at least one identifier
- An identifier can occur at most once in a pattern (*linearity*)

```
-  val t = (("datalogi", true), 25 ) ;
   val t = (("datalogi", true), 25) :
                 (string * bool) * int

-  val ( p as (name, b) , age ) = t ;
   val p = ("datalogi", true) : string * bool
   val name = "datalogi" : string
   val b = true : bool
   val age = 25 : int
```

## *Case analysis with* **case of**

Example: (pinkFloyd.sml)

```
fun albumTitle num =
   case num of
         1 => "The Piper at the Gates of Dawn"
    |    2 => "A Saucerful of Secrets"
    |    3 => "More"
    |    4 => "Ummagumma"
    |    5 => "Atom Heart Mother"
    |    6 => "Meddle"
    |    7 => "Obscured By Clouds"
    |    8 => "The Dark Side of the Moon"
    |    9 => "Wish You Were Here"
    |   10 => "Animals"
    |   11 => "The Wall"
    |   12 => "The Final Cut"
    |   13 => "A Momentary Lapse of Reason"
    |   14 => "Division Bell"
```

```
-  use "pinkFloyd.sml" ;
   ! Warning: pattern matching is not exhaustive
   val albumTitle = fn : int -> string

-  albumTitle 9 ;
   val it = "Wish You Were Here" : string

-  albumTitle 15 ;
   ! Uncaught exception: Match
```

General form:

**case** $Expr$ **of**
$Pat_1$ => $Expr_1$
| $Pat_2$ => $Expr_2$
| ...
| $Pat_n$ => $Expr_n$

- **case** ... **of** ... is an expression
- $Expr_1, \ldots, Expr_n$ must be of the *same* type
- $Expr, Pat_1, \ldots, Pat_n$ must be of the *same* type
- If the patterns are not exhaustive over their type, then there is an ML *warning* at the declaration
- If none of the patterns is applicable during an evaluation, then there is an ML pattern-matching *exception*
- The patterns need *not* be mutually exclusive: If several patterns are applicable, then ML selects the *first* applicable pattern
- If $Pat_i$ is selected, then *only* $Expr_i$ is evaluated
- Can **if** ... **then** ... **else** ... be expressed via **case** ... **of** ...?

```
fun sum a b =
  case a + b of
      0 => "zero"
    | 1 => "one"
    | 2 => "two"
    | n => if n<10 then "a lot" else "really a lot"
```

## *Case analysis with* **fun**

Example: `(pinkFloyd.sml)`

```
fun   lastAppearance "Syd Barrett"   =  2
 |    lastAppearance "Roger Waters" = 12
 |    lastAppearance x = ~1
```

General form:

**fun** $f$ $Pat_1$ = $Expr_1$
 | $f$ $Pat_2$ = $Expr_2$
 | ...
 | $f$ $Pat_n$ = $Expr_n$

## *Case analysis with* **fn**

General form:

**fn** $Pat_1$ => $Expr_1$
 | $Pat_2$ => $Expr_2$
 | ...
 | $Pat_n$ => $Expr_n$

Show that
**if** $BExpr$ **then** $Expr_1$ **else** $Expr_2$
is equivalent to
(**fn** true => $Expr_1$ | false => $Expr_2$) ($BExpr$)

# 2.11. Local declarations

*Local declarations in an expression*

**function** fraction (n,d)
TYPE: int $*$ int $\longrightarrow$ int $*$ int
PRE: d $\neq 0$
POST: $(n', d')$ such that $\frac{n'}{d'}$ is an irreducible fraction equal to $\frac{n}{d}$

Without a local declaration:

```
fun fraction (n,d) =
    ( n div gcd (n,d) , d div gcd (n,d) )
```

Recomputation of the greatest common divisor gcd (n,d)

With a local declaration: `(fraction.sml)`

```
fun fraction (n,d) =
    let val k = gcd (n,d)
    in
        ( n div k , d div k )
    end
```

Notice that the identifier k is *local* to the expression after **in** :

- Its binding exists only during the evaluation of this expression
- All other declarations of k are hidden during the evaluation of this expression

Another example:
Computation of the price of a sheet of length long and width wide, at the cost of unitPrice per square meter.
A discount of 5% is offered for every sheet whose price exceeds 250 euros: `(discount.sml)`

```
fun discount unitPrice (long,wide) =
    let val price = long * wide * unitPrice
    in
        if  price < 250.0 then price
        else price * 0.95
    end
```

- No recomputations
- Sharing of intermediate values

A last example:
Local *function* declaration in an expression: `(leapYear.sml)`

```
fun leapYear year =
     let fun isDivisible (a,b) = (a mod b) = 0
     in
         isDivisible (year,4)  andalso
         (not (isDivisible (year,100))  orelse  isDivisible (year,400))
     end
```

## *Local declarations in a declaration*

Another form for the function leapYear: `(leapYear.sml)`

```
local
   fun isDivisible (a,b) = (a mod b) = 0
in
fun leapYear2 year =
   isDivisible (year,4) andalso
   (not (isDivisible (year,100)) orelse isDivisible (year,400))
end
```

- The function isDivisible is *local* to the function leapYear2
- Better *modularity*:
  It is irrelevant whether isDivisible already exists or not

## *Differences between the two kinds of local declaration*

- **local** *Declarations* **in** *Declarations* **end**
  - Local to one or more *declarations*
  - Clearer structure, less nesting
  - Impossible confusion between the names of the arguments
  - Impossible usage in the local declaration of the values of the arguments of the principal function
- **let** *Declarations* **in** *Expression* **end**
  - Local to an *expression*
  - More nested structure
  - Possible confusion between the names of the arguments
  - Possible usage in the local declaration of the values of the arguments of the principal function

# 2.12. New operators

*Declaration of a new infix operator*

It is possible to declare new infix operators:

```
-   fun xor (p, q) =
=       (p orelse q) andalso not (p andalso q) ;
  val xor = fn : bool * bool -> bool
-   xor (true , true ) ;
  val it = false : bool
```

To write **true** xor **true** , give the following *directive*:

```
-   infix 2 xor ;
-   true xor true ;
  val it = false : bool
```

- **infix** $n$ $id$ , where $n$ is the precedence level of operator $id$
- Association to the left by default
- Association to the right with **infixr** $n$ $id$
- Possibility to return to the prefix form with **nonfix** $id$

*Using an infix operator as a prefix function*

```
-   (op xor) (true , true ) ;
  val it = false : bool
-   (op +) (3, 4) ;
  val it = 7 : int
```

# 2.13. Recursive functions

## Example: Factorial

*Specification*

**function** fact n
TYPE: int $\longrightarrow$ int
PRE: n $\geq$ 0
POST: n!

*Construction*

Error case: n < 0 : produce an error message
Base case: n = 0 : the result is 1
General case: n > 0 : the result is n ∗ fact (n−1)

*ML program* (fact.sml)

```
fun fact n =
    if n < 0 then error "fact: negative argument"
    else if n = 0 then 1
    else n * fact (n-1)
val rec fact = fn n =>
    if n < 0 then error "fact: negative argument"
    else if n = 0 then 1
    else n * fact (n-1)
```

# 2.14. Side effects

Like most functional languages,
ML has some functions with side effects:

- Input / output
- Variables (in the imperative-programming sense)
- Explicit references
- Tables (in the imperative-programming sense)
- Imperative-programming-style control structures (sequence, iteration, ...)

In these lectures:
Limitation to the printing of results and the loading of files

*The* **print** *function*

Type: **print** string → unit
Side effect: The argument of **print** is printed on the screen

*Example*

```
-  fun welcome msg = print (msg ^ "\n") ;
   val welcome = fn : string -> unit

-  welcome "hello" ;
   hello
   val it = () : unit
```

## *Sequential composition*

Sequential composition is necessary when, for example, one wants to print intermediate results: `(relError.sml)`

```
fun relError a b =
    let val diff = abs (a-b)
    in
        ( print (Real.toString diff) ;
          print "\n" ;
          diff / a )
    end
```

- Sequential composition is an *expression* of the form

$$( Expr_1 \,;\, Expr_2 \,;\, \ldots \,;\, Expr_n )$$

- The value of this expression is the value of $Expr_n$

## *The* **use** *function*

Loading and evaluation of the content of a file named $f$ with ML expressions: via **use** "$f$" ;

This allows the declaration of functions in a file

This function is primarily used in *interactive mode*

# 2.15. Exception declarations

Execution can be interrupted immediately upon an error

*Example*

```
exception errorDiv
fun safeDiv a b =
    if b = 0 then raise errorDiv
    else a div b

-   45 * (safeDiv 23 0) + 12 ;
  Uncaught exception: errorDiv
```

*Error function* (`error.sml`)

In these lectures, to simplify matters,
we will use a single function for treating all errors:

**function** error msg
TYPE: string $\longrightarrow$ (to be completed later)
SIDE-EFFECT: displays msg to the screen and halts the execution

```
exception StopError
fun error (msg:string) =
    ( print msg ; print "\n" ; raise StopError )
```

## *Examples*

(safeDiv.sml)

```
fun safeDiv a b =
    if b=0 then error "safeDiv: division by 0"
    else a div b
```

```
-  45 * (safeDiv 23 0) + 12 ;
   safeDiv: division by 0
   Uncaught exception: StopError
```

(log.sml)

```
fun logBis base x =
    if base <= 0 then error "logBis: non-positive base"
    else Math.ln x / Math.ln (real base)
```

```
-  logBis ~2 12.3 ;
   logBis: non-positive base
   Uncaught exception: StopError
-  val logTwo = logBis ~2 ;
   val logTwo = fn : real -> real
-  logTwo 16.0 ;
   logBis: non-positive base
   Uncaught exception: StopError
```

Use an anonymous function to directly verify the base (log.sml) :

```
fun logTer base =
    if base <= 0 then error "logTer: non-positive base"
    else fn x => Math.ln x / Math.ln (real base)

-  val logTwo = logTer ~2 ;
   logTer: non-positive base
   Uncaught exception: StopError
```

- What is the type of function logTer?
- Reduce the expression logTer 2 16.0

# 2.16. Functional languages vs. imperative languages

*Example: greatest common divisor of natural numbers*

We know from Euclid that:

$$\begin{aligned} gcd(0,n) &= n \quad &\text{if } n > 0 \\ gcd(m,n) &= gcd(n \bmod m, m) \quad &\text{if } m > 0 \end{aligned}$$

*Pascal program*

```
function gcd(m, n : integer) : integer;
{# PRE: m, n ≥ 0 and m + n > 0
   POST: gcd = the greatest common divisor of m, n #}
var a, b, prevA : integer ;
begin
   a := m ; b := n ;
   {# INVARIANT: gcd(m,n) = gcd(a,b) #}
   while a <> 0 do
   begin
      prevA := a ;
      a := a mod b ;
      b := prevA
   end ;
   gcd := b
end
```

## *Features of imperative programs*

- Close to the hardware
  - Sequence of instructions
  - Modification of variables (memory cells)
  - Test of variables (memory cells)
  - Transformation of states (automata)
- Construction of programs
  - Describe what has to be computed
  - Organise the sequence of computations into steps
  - Organise the variables
- Correctness
  - Specifications by pre/post-conditions
  - Loop invariants
  - Symbolic execution
- Expressions
  `f(z) + x / 2` can be different from `x / 2 + f(z)`
  namely when `f` modifies the value of `x` (by side effect)
- Variables
  The assignment `x := x + 1`
  modifies a memory cell as a side effect

*Specification*

**function** gcd (m, n)
TYPE: int $*$ int $\longrightarrow$ int
PRE: m, n $\geq$ 0 and m + n > 0
POST: the greatest common divisor of m, n

*ML program* (gcd.sml)

```
fun gcd1 (m, n) =
    if m = 0 then n
    else gcd1 (n mod m, m)

fun   gcd2 (0, n) = n
  |   gcd2 (m, n) = gcd2 (n mod m, m)
```

*Features of functional programs*

- Execution by evaluation of expressions
- Basic tools: expressions and recursion
- Handling of *values* (rather than states)
- The expression $e_1 + e_2$ *always* has the same value as $e_2 + e_1$
- Identifiers
  - Value via a *declaration*
  - No assignment, no "modification"
- Recursion: series of values from recursive calls