Chapter 2
ML, a Functional Programming Language

(Version of 24 September 2004)

1. Expressions ............................................. 2.2
2. Value declarations ................................. 2.13
3. Function declarations ............................. 2.16
4. Type inference ...................................... 2.18
5. Anonymous functions ............................. 2.20
6. Specifications ....................................... 2.22
7. Tuples and records ................................. 2.24
8. Functions with several arguments/results ...... 2.26
9. Currying ................................................. 2.28
10. Pattern matching and case analysis .......... 2.32
11. Local declarations ................................. 2.36
12. New operators ....................................... 2.39
13. Recursive functions ............................... 2.40
14. Side effects .......................................... 2.41
15. Exception declarations ........................... 2.42
16. Functional languages vs. imperative languages 2.46
2.1. Expressions

Interacting with ML

- \(32 + 15\);
  \[
  \text{val it} = 47 : \text{int}
  \]
- \(3.12 \times 4.3\);
  \[
  \text{val it} = 13.416 : \text{real}
  \]
- \(\text{not true}\);
  \[
  \text{val it} = \text{false} : \text{bool}
  \]
- "The Good, the Bad," ^ " and the Ugly" ;
  \[
  \text{val it} = "\text{The Good, the Bad, and the Ugly}" : \text{string}
  \]
- ( size("Esra") +
  = size("Pierre") ) div 2 ;
  \[
  \text{val it} = 5 : \text{int}
  \]

- 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

```plaintext
- 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:

- `+ : int × int → int`
- `+ : real × real → real`
Integers

Syntax

- As usual, except the unary operator $-$ is represented by ~
- Example: $\sim 123$

Basic operators on the integers

<table>
<thead>
<tr>
<th>$op$</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{int}$</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>$-$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{int}$</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>$\ast$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{int}$</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>$\div$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{int}$</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>$\mod$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{int}$</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>$=$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{bool}$</td>
<td>* infix</td>
<td>4</td>
</tr>
<tr>
<td>$&lt;&gt;$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{bool}$</td>
<td>* infix</td>
<td>4</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{bool}$</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>$\leq$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{bool}$</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{bool}$</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>$\geq$</td>
<td>$\text{int} \times \text{int} \rightarrow \text{bool}$</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>$\sim$</td>
<td>$\text{int} \rightarrow \text{int}$</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>$\text{int} \rightarrow \text{int}$</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

(* 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, \(^{-}12.34\), \(^{-}34E2\), 4.57E3

Basic operators on the reals

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>real × real → real</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>−</td>
<td>real × real → real</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>real × real → real</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>/</td>
<td>real × real → real</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>~</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>Math.sqrt</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>Math.In</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

(* 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
Basic operators on the characters and strings

Let ‘strchar × strchar’ be ‘char × char’ or ‘string × string’

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>strchar × strchar → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>strchar × strchar → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;</td>
<td>strchar × strchar → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>strchar × strchar → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>strchar × strchar → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>strchar × strchar → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>^</td>
<td>string × string → string</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>size</td>
<td>string → int</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

(* 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

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>andalso</td>
<td>bool × bool → bool</td>
<td>infix</td>
<td>3</td>
</tr>
<tr>
<td>orelse</td>
<td>bool × bool → bool</td>
<td>infix</td>
<td>2</td>
</tr>
<tr>
<td>not</td>
<td>bool → bool</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>bool × bool → bool</td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>bool × bool → bool</td>
<td>*</td>
<td>4</td>
</tr>
</tbody>
</table>

(* the exact type will be defined later)

Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A andalso B</th>
<th>A orelse B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
• 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.In(12.4) * 3.4 > 12.0);
  val it = true : bool
```

The second operand, namely *(Math.In(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

\[
\text{\textbf{op}} \quad : \quad \text{\textbf{type}}
\]

\begin{align*}
\text{real} & : \quad \text{int} \rightarrow \text{real} \\
\text{ceil} & : \quad \text{real} \rightarrow \text{int} \\
\text{floor} & : \quad \text{real} \rightarrow \text{int} \\
\text{round} & : \quad \text{real} \rightarrow \text{int} \\
\text{trunc} & : \quad \text{real} \rightarrow \text{int}
\end{align*}

\begin{itemize}
  \item \texttt{real(2) + 3.5 ;}
    \hspace{1cm} \texttt{val it = 5.5 : real}
  \item \texttt{ceil(23.65) ;}
    \hspace{1cm} \texttt{val it = 24 : int}
  \item \texttt{ceil(~23.65) ;}
    \hspace{1cm} \texttt{val it = ~23 : int}
  \item \texttt{floor(23.65) ;}
    \hspace{1cm} \texttt{val it = 23 : int}
  \item \texttt{floor(~23.65) ;}
    \hspace{1cm} \texttt{val it = ~24 : int}
  \item \texttt{round(23.65) ;}
    \hspace{1cm} \texttt{val it = 24 : int}
  \item \texttt{round(23.5) ;}
    \hspace{1cm} \texttt{val it = 24 : int}
  \item \texttt{round(22.5) ;}
    \hspace{1cm} \texttt{val it = 22 : int}
\end{itemize}
- \texttt{trunc}(23.65) ;
  \texttt{val it = 23 : int}

- \texttt{trunc}('23.65) ;
  \texttt{val it = ~23 : int}

\begin{align*}
\text{op} & : \text{type} \\
\text{chr} & : \text{int} \rightarrow \text{char} \\
\text{ord} & : \text{char} \rightarrow \text{int} \\
\text{str} & : \text{char} \rightarrow \text{string}
\end{align*}

- \texttt{chr}(97) ;
  \texttt{val it = \#"a" : char}

- \texttt{ord}(\#"a") ;
  \texttt{val it = 97 : int}

- \texttt{str}(\#"a") ;
  \texttt{val it = "a" : string}

Conversions are done according to the ASCII code
Evaluation of expressions

**Reduction**

\[ 3 + 4 \times 2 < 5 \times 2 \]
\[ \rightsquigarrow 3 + 8 < 5 \times 2 \]
\[ \rightsquigarrow 11 < 5 \times 2 \]
\[ \rightsquigarrow 11 < 10 \]
\[ \rightsquigarrow \text{false} \]

- 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 \( E_1 \ \text{op} \ E_2 \)

1. Reduction of the expression \( E_1 \): \( E_1 \rightsquigarrow \ldots \rightsquigarrow N_1 \)
2. Reduction of the expression \( E_2 \): \( E_2 \rightsquigarrow \ldots \rightsquigarrow N_2 \) unless \( \text{op} \) is lazy and \( N_1 \) is such that \( E_2 \) need not be reduced
3. Application of the operator \( \text{op} \) 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

\[ \leadsto \text{if true then } 4.1 + 2.3 \text{ else } 2.1 / 0.0 \]

\[ \leadsto 4.1 + 2.3 \]

\[ \leadsto 6.4 \]

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

Examples

- \texttt{val pi = 3.14159 ;}
  \texttt{val pi = 3.14159 : real}

- \texttt{val twoPi = 2.0 * pi ;}
  \texttt{val twoPi = 6.28318 : real}

- \texttt{twoPi * 5.3 ;}
  \texttt{val it = 33.300854 : real}

- \texttt{it / 2.0 ;}
  \texttt{val it = 16.650427 : real}

- \texttt{val &@!+<% = "bizarre, no?!" ;}
  \texttt{val &@!+<% = "bizarre, no?!" : string}

Identifiers

- Alphanumeric identifiers
- Symbolic identifiers
  made from \(+/-/\ast/\le = ! @ \# $ % ^ \& \~ \| ? :\)
- Do not mix alphanumeric and symbolic characters
- The identifier \texttt{it} always has the result of the last unidentified expression evaluated by the interpreter
- Attention: \texttt{3 +\~ 2} is different from \texttt{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

- \texttt{fun signSquare( x : int ) : int = abs(x) \times x ;}
\begin{verbatim}
val signSquare = fn : int \to int
\end{verbatim}
- \texttt{signSquare(\textasciitilde3) ;}
\begin{verbatim}
val it = \textasciitilde9 : int
\end{verbatim}

• The used function \texttt{abs} must have been declared beforehand
• Possibility of simultaneous declarations:

- \texttt{fun signSquare( x : int ) : int = abs(x) \times x}
= \texttt{and abs( x : int ) : int = if x \geq 0 then x else \textasciitilde x ;}
\begin{verbatim}
val signSquare = fn : int \to int
val abs = fn : int \to int
\end{verbatim}

Evaluation: reduction

\begin{verbatim}
signSquare(3-6)
\mapsto signSquare(\textasciitilde3)
\mapsto abs(\textasciitilde3) \times \textasciitilde3
\mapsto (if \textasciitilde3 \geq 0 then \textasciitilde3 else \textasciitilde(\textasciitilde3)) \times \textasciitilde3
\mapsto (if false then \textasciitilde3 else \textasciitilde(\textasciitilde3)) \times \textasciitilde3
\mapsto \textasciitilde(\textasciitilde3) \times \textasciitilde3
\mapsto 3 \times \textasciitilde3
\mapsto \textasciitilde9
\end{verbatim}
2.4. Type inference

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

\textbf{Example}

\begin{verbatim}
  fun abs( x ) =
      if x >= 0 then x else ~ x ;
  val abs = fn : int -> int
\end{verbatim}

From \( x \geq 0 \), the ML interpreter infers that \( x \) must necessarily be of type \texttt{int} because the type \texttt{int} of 0 is recognised from the syntax; hence the result of \texttt{abs} must be of type \texttt{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 ;

  val square = fn : real -> real
```

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:

- \[
\text{fun double(} \ x \ \text{)} = 2 \times x \ ; \ \\
\text{val double} = \text{fn : int} \rightarrow \text{int}
\]

- \[
\text{val double} = \text{fn} \ x \Rightarrow 2 \times x \ ; \\
\text{val double} = \text{fn : int} \rightarrow \text{int}
\]

- \[
\text{double} ; \\
\text{val it} = \text{fn : int} \rightarrow \text{int}
\]

- \[
\text{double(3) } ; \\
\text{val it} = 6 : \text{int}
\]

- \[
\text{fn} \ x \Rightarrow 2 \times x ; \\
\text{val it} = \text{fn : int} \rightarrow \text{int}
\]

- \[
(\text{fn} \ x \Rightarrow 2 \times x)(3) ; \\
\text{val it} = 6 : \text{int}
\]

The forms \textbf{fun Name Arg = Def} and \textbf{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
\[ \leadsto (\text{fn } x \Rightarrow 2 \times x)(3) + 4 \]
\[ \leadsto (2 \times 3) + 4 \]
\[ \leadsto 6 + 4 \]
\[ \leadsto 10 \]

- 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 \( E_1 \): \[ E_1 \leadsto \ldots \leadsto N_1 \]
   \( N_1 \) must be of the form \( \text{fn } Arg \Rightarrow Def \)
2. Reduction of the expression \( E_2 \): \[ E_2 \leadsto \ldots \leadsto N_2 \]
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 → int
PRE: n ≥ 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 → int
PRE: (none)
POST: the largest integer m such that m ≤ n
EXAMPLES: floor(23.65) = 23, floor(\textasciitilde23.65) = \textasciitilde24
COUNTER-EXAMPLE: floor(\textasciitilde23.65) \neq \textasciitilde23
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, \ldots
- 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 \( i \)th component of a tuple
- It is possible to have tuples of tuples
- The value () is the only 0-tuple, and it has type \textit{unit}
- The expression \((e)\) is equivalent to \( e \), hence \textit{not} a 1-tuple!
  
  Example: \texttt{sum(n)} can also be written as \texttt{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

- \(\#2(\#1\ \text{bigTuple}) + \#1(\#3\ \text{bigTuple})\)
  
  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

- \{\text{course} = "FP", \text{year} = 2\} ;
  \begin{verbatim}
  val it = \{\text{course} = "FP", \text{year} = 2\} : \\
  \{\text{course} : \text{string}, \text{year} : \text{int}\}
  \end{verbatim}
- Selector \#label returns the value of the component identified 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"\} ;
    \begin{verbatim}
    val it = 1 : \text{int}
    \end{verbatim}
  - \{a=1, b="xyz"\} = \{b="xyz", a=1\} ;
    \begin{verbatim}
    val it = true : \text{bool}
    \end{verbatim}
  - \(1, \text{"xyz"}) = (\text{"xyz"}, 1) ;
    \begin{verbatim}
    ! (1, "xyz") = ("xyz", 1);
    ! Type clash: expression of type string
    ! cannot have type \text{int}
    \end{verbatim}
  - \{1=1, 2="xyz"\} = (1, "xyz") ;
    \begin{verbatim}
    val it = true : \text{bool}
    \end{verbatim}
2.8. Functions with several arguments/results

In ML, a function always has:

- a unique argument
- a unique result

“Multiple-argument” functions

```ml
- 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)

```ml
- 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
There is equivalence of the types of the following functions:

\[ f : A \times B \to C \]

\[ g : A \to (B \to C') \]

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 \( B \to C \)
  
  \( g a \) 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:

\[ g : A_1 \times A_2 \times \cdots \times A_n \rightarrow C \]

\[ \Downarrow \]

\[ g : A_1 \rightarrow (A_2 \rightarrow \cdots \rightarrow (A_n \rightarrow C)) \]

\[ g : A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow C \]

The symbol \( \rightarrow \) 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 → real → real
PRE: base > 0
POST: log\textit{base} x

fun log base x = Math.In x / Math.In (real base)

- \texttt{log 2 12.3 ;}
  \texttt{val it = 3.62058641045 : real}

- \texttt{fun logTwo x = log 2 x ;}
  \texttt{val logTwo = fn : real → real}

- \texttt{logTwo 16.0 ;}
  \texttt{val it = 4.0 : real}

Reduction

log 2 16.0
\rightarrow (\texttt{fn base ⇒ (fn x ⇒ Math.In x / Math.In (real base))) 2 16.0}
\rightarrow (\texttt{fn x ⇒ Math.In x / Math.In (real 2)}) 16.0
\rightarrow Math.In 16.0 / Math.In (real 2)
\rightarrow 2.77258872224 / Math.In (real 2)
\rightarrow 2.77258872224 / Math.In 2.0
\rightarrow 2.77258872224 / 0.69314718056
\rightarrow 4.0
The currying of \texttt{log} is irrelevant here

\begin{itemize}
\item \texttt{log 2 12.3 ;}
  \texttt{val it = 3.62058641045 : real}
\item \texttt{val logTwoBis = log 2 ;}
  \texttt{val logTwoBis = \texttt{fn} : real \rightarrow real}
\item \texttt{logTwoBis 16.0 ;}
  \texttt{val it = 4.0 : real}
\end{itemize}

\texttt{log 2}
\begin{itemize}
\item \texttt{(fn base \rightarrow (fn x \rightarrow Math.In x / Math.In (real base))) 2}
\item \texttt{(fn x \rightarrow Math.In x / Math.In (real 2))}
\end{itemize}

\texttt{logTwoBis 16.0}
\begin{itemize}
\item \texttt{(fn x \rightarrow Math.In x / Math.In (real 2)) 16.0}
\item \texttt{Math.In 16.0 / Math.In (real 2)}
\item \texttt{...}
\end{itemize}

The currying of \texttt{log} is essential here

Why can \texttt{logTwoBis} not be declared with \texttt{fun} rather than \texttt{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)

```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
```

© P. Flener/IT Dept/Uppsala Univ. FP 2.33
General form:

```ml
case Expr of
    Pat_1 => Expr_1
| Pat_2 => Expr_2
| ... 
| Pat_n => Expr_n
```

- `case ... of ...` is an expression
- `Expr_1, ..., Expr_n` must be of the same type
- `Expr, Pat_1, ..., 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 ...`?

```ml
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 \texttt{fun}

Example: (\texttt{pinkFloyd.sml})

\begin{verbatim}
fun lastAppearance "Syd Barrett"  = 2
| lastAppearance "Roger Waters" = 12
| lastAppearance x = ~1
\end{verbatim}

General form:

\begin{verbatim}
fun \texttt{f Pat}_1 = \texttt{Expr}_1
| \texttt{f Pat}_2 = \texttt{Expr}_2
| \ldots
| \texttt{f Pat}_n = \texttt{Expr}_n
\end{verbatim}

Case analysis with \texttt{fn}

General form:

\begin{verbatim}
fn \texttt{Pat}_1 \Rightarrow \texttt{Expr}_1
| \texttt{Pat}_2 \Rightarrow \texttt{Expr}_2
| \ldots
| \texttt{Pat}_n \Rightarrow \texttt{Expr}_n
\end{verbatim}

Show that

\begin{verbatim}
\texttt{if BExpr then Expr}_1 \texttt{ else Expr}_2
\end{verbatim}

is equivalent to

\begin{verbatim}
(\texttt{fn true} \Rightarrow \texttt{Expr}_1 | \texttt{false} \Rightarrow \texttt{Expr}_2) (BExpr)
\end{verbatim}
2.11. Local declarations

Local declarations in an expression

function fraction (n,d)
  TYPE: int * int → int * int
  PRE: d ≠ 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 \texttt{long} and width \texttt{wide}, at the cost of \texttt{unitPrice} per square meter.
A discount of 5\% is offered for every sheet whose price exceeds 250 euros: (\texttt{discount.sml})

\begin{verbatim}
fun discount unitPrice (long,wide) = 
  let val price = long * wide * unitPrice 
  in 
    if price < 250.0 then price 
    else price * 0.95 
  end
\end{verbatim}

- No recomputations
- Sharing of intermediate values

A last example:
Local \textit{function} declaration in an expression: (\texttt{leapYear.sml})

\begin{verbatim}
fun leapYear year = 
  let fun isDivisible (a,b) = (a mod b) = 0 
  in 
    isDivisible (year,4) \texttt{andalso} 
    (\texttt{not} (isDivisible (year,100)) \texttt{orelse} isDivisible (year,400)) 
  end
\end{verbatim}
Local declarations in a declaration

Another form for the function \texttt{leapYear}: (\texttt{leapYear.sml})

\begin{verbatim}
local
  fun isDivisible (a,b) = (a mod b) = 0
in
  fun leapYear2 year = 
    isDivisible (year,4) \textbf{andalso}
    (not (isDivisible (year,100)) \textbf{orelse} isDivisible (year,400))
end
\end{verbatim}

- The function \texttt{isDivisible} is local to the function \texttt{leapYear2}
- Better modularity:
  It is irrelevant whether \texttt{isDivisible} already exists or not

Differences between the two kinds of local declaration

- \textbf{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

- \textbf{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:

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

val it = false : bool
```

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

```ml
infix 2 xor ;
val 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

```ml
(val xor (true, true));
val it = false : bool

(val op +) (3, 4); 
val it = 7 : int
```
2.13. Recursive functions

Example: Factorial

Specification

```ml
function fact n
TYPE: int → int
PRE: n ≥ 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 \( n \times \text{fact}(n-1) \)

ML program (fact.sml)

```ml
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)

```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

  $$ ( E_{\text{expr}_1} ; E_{\text{expr}_2} ; \ldots ; E_{\text{expr}_n} ) $$

- The value of this expression is the value of $E_{\text{expr}_n}$

The use function

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

```sml
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

```ml
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:

```ml
function error msg
TYPE: string → (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.In x / Math.In (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):

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

\[
\begin{align*}
gcd(0, n) &= n & \text{if } n > 0 \\
gcd(m, n) &= gcd(n \mod m, m) & \text{if } m > 0
\end{align*}
\]

Pascal program

```
function gcd(m, n : integer) : integer;
{ # PRE: m, n \geq 0 and m + n > 0
   POST: gcd = the greatest common divisor of m, n  # }
var a, b, prevA : integer ;
begin
    a := m ; b := n ;
    { # INARIANT: 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**

```ml
function gcd (m, n) TYPE: int * int → int
PRE: m, n ≥ 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