Review of ML

ML ("Meta Language") is a *functional programming language* based on the concept of evaluation via substitution and replacement.

ML was originally introduced in the 1970's as part of a theorem proving system, intended for describing and implementing proof strategies. Various versions of ML are currently available, check out the "supplementary material" on the course webpage for further information.

The primary mode of computation in ML, as in other functional languages, is the use of the definition and application of functions.

The basic cycle of ML activity consists of three parts:

- *read* input from the user,
- *evaluate* it, and
- *print* the computed value (or an error message).

Interacting with ML

A simple example illustrates this:

```
- val P1 = [2,3];
val P1 = [2,3] :  int list
```

The first line contains the ML prompt, followed by an expression (typed in by the user) and ended by a semi-colon.

The second line is ML's response and shows the value of the input expression and its type.

ML provides a number of built-in operators and data structures such as integers and reals with the standard arithmetic operators, or lists and associated list operations.

```
- 3+2;
val it = 5 :  int
- 5.0/3.0;
val it = 1.66666666667 :  real
5 div 3;
val it = 1 :  int
- 5 mod 3;
val it = 2 :  int
```

Values and Types

ML is a *strongly typed* language in that all (well-formed) expressions have a (well-defined) type that can be determined by examining the expression.

Types are determined as part of the evaluation process. Some (built-in) operators are "overloaded":

```
- 3.0+2.0;
val it = 5.0 : real
```

In other words, the symbol **+** may denote a function on the integers or on the reals. But there is no implicit conversion from values of one type to values of another type, say between integers and reals:

```
- 3+2.0;
Error: operator and operand don't agree
operator domain: int * int
operand: int * real
in expression: 3 + 2.0
```

Boolean Operations in ML

ML uses the constants `true` and `false` to denote the Boolean values. The common logical operations can be performed via the functions `not` (negation), `andalso` (conjunction), and `orelse` (disjunction).

```
- not(true);
val it = false :  bool
- true andalso false;
val it = false :  bool
```

The conditional, or if-then-else, operator takes three arguments, the first of which must yield a Boolean value:

```
- if 1>2 then 1 else 2;
val it = 2 :  int
```

The second and the third argument, which follow `then` and `else`, respectively, can be of any type, but they must be of the *same* type.

```
- if 1>2 then 1 else 2.0;
Error:  types of rules don't agree
earlier rule(s):  bool -> int
this rule:  bool -> real
in rule:  false => 2.0
```

(In ML if-then-else is actually a short-hand for a specific instance of a more general so-called case expression.)

List Operations in ML

The basic list operations provided by ML are:

- a binary cons operator which takes a first argument and adds it at the beginning of its second argument,

  ```
  - 1::[2,3,5];
  val it = [1,2,3,5] :  int list
  ```

- unary operators which return the first element of a list and the list of all elements but the first, respectively,

  ```
  - hd[2,3,5];
  val it = 2 :  int
  - tl[2,3,5];
  val it = [3,5] :  int list
  ```

- and a binary operator for concatenation of lists,

  ```
  - [2,3,5]@[7];
  val it = [2,3,5,7] :  int list
  ```

One of the basic operations characterizing arrays, namely the extraction of the n-th term in the sequence, is not provided for lists. Such a function can be defined, of course, but requires traversing part of the list.

Defining Functions in ML

The general form of a function definition in ML is

fun $\langle$identifier$\rangle$ ($\langle$parameters$\rangle$) = $\langle$expression$\rangle$;

For example,

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

defines a function on reals:

```
- square(2.0);
val it = 4.0 :  real
```

If no type is specified for x, then ML will use the default type, integer.

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

If a function is applied to an argument of the wrong type, ML produces an error message:

```
- square[2];
Error:  operator and operand don't agree
operator domain:  int
operand:  int list
in expression:  square (2 ::  nil)
```

Recursive Definitions

The extensive use of recursive definitions is a distinguishing characteristic of functional programs as functional languages strongly encourage recursion as a structuring mechanism in preference to iterative constructs such as while-loops.

To illustrate recursion we give a function that produces the reverse of a given list.

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

The else-part contains the general recursive case of the definition, while the then-part provides the base case for the recursion.

```
- reverse[1,2,3];
val it = [3,2,1] :  int list
```

Be sure to define a base case (or cases) for a recursive definition, as otherwise the evaluation of function applications may not terminate.

Polymorphism

The function `reverse` is an example of a *polymorphic* function: it can be applied to arguments of type `'a list`, that is, to lists of values of any arbitrary type; e.g., lists of integers, lists of reals, or lists of lists of integers.

```
- reverse[[1],[2,2],[3,3,3]];
val it = [[3,3,3],[2,2],[1]] :  int list list
```

Another example of a polymorphic function is the identity function:

```
- fun identity(x) = x;
val identity = fn :  'a -> 'a
```

Here again `'a` denotes a type variable which can be instantiated in different ways when the identity function is applied to specific arguments. The type `'a -> 'a` can be thought of as a *type schema*.

```
- identity(5);
val it = 5 :  int
- identity(7.5);
val it = 7.5 :  real
- identity(5)+floor(identity(7.5));
val it = 12 :  int
```

Note that in the last example the function `identity` is applied both to an integer and to a real number.

Restrictions to Polymorphism in ML

The list operators and the equality operator **=** are examples of ML operators that allow polymorphism. But ML also contains operators that restrict polymorphism.

Some operators require arguments, and produce results, of a specific type. These include the Boolean operators `not`, `andalso`, and `orelse` and some of the arithmetical operators, such as `/`, `div` and `mod`.

There are also seen operators that may be applied to values of different types, though the type of each argument must be known from inspection of the function. (This may require that the type of an argument be explicitly specified.) These functions include `+`, `*` and `<`.

ML is a strongly typed language, in which it is possible to determine the type of each correct expression by examining it. If a type cannot be determined, the expression is by definition incorrect. The algorithm used by ML for deducing types is complex and beyond the scope of this course.