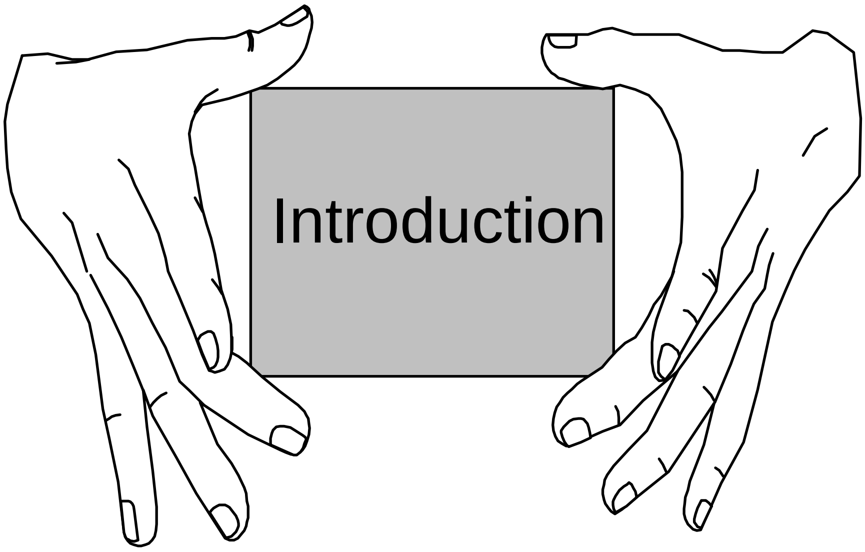
Standard ML



ML Introduction.1

Recommended Text Book

- Includes:
  - Most of the examples
  - Most of the questions
  - Most of the answers...

ML Introduction.2

Loading and Saving

- Interpreter
- First prompt (-) and secondary prompt (=)
- Need ; after each expression/definition
- Loading ML source text from a file
  - Create a file named “`myfile.sml`”
  - Either start ML
    - and use the function `use: string -> unit`

      `- use “c:\\myfile.sml”;`

    `Don’t forget the double ‘\’ in the path!`
  - Or redirect the input and output
    - **`C:\ sml < myfile.sml > output`**

ML Introduction.3

A simple tutorial

- ML is usually used as interpreter (a compiler is also available)
- Expressions followed by a semicolon yield a response

```
- 2+2;
val it = 4 : int
```

- Doing simple arithmetic

```
- 3.2 - 2.3;
val it = 0.9 : real
- Math.sqrt(2.0);
val it = 1.414213562 : real
```

ML Introduction.4

Notes

Declaring Constants

- Naming constants
  - val seconds = 60;
  val seconds = 60 : int
- minutes per hour
  - val minutes = 60;
  val minutes = 60 : int
- hours per day
  - val hours = 24;
  val hours = 24 : int
- Using names in expressions
  - seconds * minutes * hours;
  val it = 86400 : int

The identifier ‘*it*’

- By referring to *it,* one can use the last value
  - it div 24;
  val it = 3600 : int
- Any previous value of *it* is lost unless saved
  - val secsinhour = it;
  val secsinhour = 3600 : int;
- Underscores can be used in names for readability
  - val secs_in_hour = seconds * minutes;
  val secs_in_hour = 3600 : int

Legal Names - Alphabetic Names

- Alphabetic name
  - Begins with a letter
  - Then followed by letters, digits, underscore, or single quotes
    - Examples:
      - **x**
      - **UB40**
      - **Hamlet_Prince_of_Denmark**
      - **h’’3_H**
      - **or_any_other_name_that_is_as_long_as_you_like**
  - Case of letters matters
  - ML was designed by Mathematicians who like primes
      - **x, x’, x’’, x’’’**

ML Introduction.7

Legal Names - Symbolic Names

- Permitted over the characters:
  ```
  ! % & $ # + - * /  : < = > ? @ \ ~ ‘ ^ |
  ```
- May be as long as you like:
  ```
  ---->
  $^$^$^$
  !!?@**??!!
  :_|==>->#
  ```
- Should not be one of the ML reserved special syntax:
  ```
  :  _  |  =  =>  ->  #
  ```
- Allowed whenever an alphabetic name is:
  ```
  - val +-+-+ = 1415;
  val +-+-+ = 1415 : int
  ```

ML Introduction.8

ML's keywords

`abstype and andalso as case datatype do else end eqtype exception fn fun functor handle if in include infix infixr let local nonfix of op open orelse raise rec sharing sig signature struct structure then type val while with withtype`

- Avoid ML's keywords when choosing names
- Especially watch out from the short ones:
  `as fn if in of op`

ML Introduction.9

ML Primitive Types

int, real, string, char, bool, unit

ML Introduction.10

Integer Types

- Constants
  - sequence of digits
    - **`0`**
    - **`01234`**
  - **~** for a unary minus sign
    - **`~23`**
    - **`~85601435654638`**
- Infix operations:
  `+ - * div mod`
- Conventional precedence
  `(((m * n) * l) - (m div j)) + j`
    - parenthesis can be dropped without change of meaning.

ML Introduction.11

Real types

- Constants
  - decimal point
    - **`0.01`**
    - **`2.718281828`**
  - E notation
    - **`7E~5`**
    - **`~1.2E12`**
    - **`~123.4E~2`** is the same as **`~1.234`**
  - **~** for unary minus
- Infix operators
  `+ - * /`
- Functions
  - **`floor(r)`** converts real to int, **`real(i)`** converts int to real
  - **`sqrt, sin, cos, tan, exp, ln`** all of type *real -> real*
  - All need the **Math** prefix: **`Math.sqrt`**, **`Math.sin`**
  - Infix operators have lower precedence.

ML Introduction.12

Strings

- Constants are written in double quotes

```
- "ML is the best";
val it = "ML is the best" : string
```

- Special characters **\n \t \" **
- Concatenation

```
- "Standard" ^ " ML";
val it = "Standard ML" : string
```

- **size** returns the number of characters

```
- size (it);
val it = 11 : int
```

size("") is 0

- ***Infix operators***

```
<, ^
```

ML Introduction.13

Characters

- Chars are distinguished from strings of length 1 by the **#** sign

```
- "0";
val it = "0" : string
- #"0";
val it = #"0" : char
```

- Converting between strings and characters using **str** and **sub**

```
- str(#"0");
val it = "0" : string
- String.sub("hello",0);
val it = #"h" : char
```

- Converting chars to and from ASCII using **ord** and **chr**

```
- ord #"0";
val it = 48 : int
- chr it;
val it = #"0" : char
```

ML Introduction.14

Notes

Boolean

- The two values are

```
- true;
val it = true : bool
- false;
val it = false : bool
```


Tuples
Cartesian Product Type

- **(x1, x2, ... ,xn)**
  - The n-tuple whose components are x1,x2,...,xn.
  - The components can be of any type, including tuples.
- Examples

```
– val a = (1.5, 6.8);
val a = (1.5, 6.8) : real * real
– (1, 1.5);
val it = (1, 1.5) : int * real
– ("str",1,true,(#"0",0.1));
val it = ("str",1,true,(#"0",0.1)) : string * int * bool * (char * real)
```

Records

- Records have components (fields) identified by name

```
- val me = { name="Ofir", age=30 };
```

val me = {age=30,name="Ofir"} :
{age:int, name:string}

- Type lists each field as `label : type`
- Enclosed in braces { ... }
- Selecting a field

```
- #name(me);
```

val it = "Ofir" : string

- Tuples can be seen as records with numbers as implicit field labels
  - `(x1,x2,...,xn)` is `{1=x1, 2=x2,..., n=xn}`

```
- #2 ("one", "two", "three");
```

val it = "two" : string

ML Introduction.17

Lists

- A list is a finite sequence of elements.
  - [3,5,9]
  - ["a", "list"]
  - []
- Elements may appear more than once
  - [3,4]
  - [4,3]
  - [3,4,3]
  - [3,3,4]
- Elements may have any type. But all elements of a list must have the same type.
  - [(1,"One"),(2,"Two")] : (int*string) list
  - [[3.1],[],[5.7, ~0.6]]: (real list) list

ML Introduction.18

Notes

Mapping - Functions

- `- fun sq(x:int) = x*x;`
  `val sq = fn : int -> int`
  - keyword **`fun`** starts the function declaration
  - ***`sq`*** is the function name
  - ***`x :int`*** is the formal parameter with type constraint
  - ***`x*x`*** is the body and it is an **expression**
  - the type of a function is printed as ***`fn`***
  - The result of the function is the result of evaluating the **expression** of the function body with the actual parameter
  - ***`int -> int`*** is the standard mathematical notation for a function type that takes a real number and returns a real number

ML Introduction.19

Applying a Function

- Simple function call
  ```
  - sq (3);
  val it = 9 : int
  ```
- When a function is called the parameter is evaluated and then passed to the function (seems obvious but it is not always the case in functional languages…)
  ```
  - sq (sq(3));
  val it = 81 : int
  ```
- The parentheses around the argument are optional
  ```
  - sq 3;
  val it = 9 : int
  ```
- Parentheses are also optional in function definitions
  ```
  - fun sq x:int = x*x;
  val sq = fn: int -> int
  ```

ML Introduction.20

Notes

Arguments and Results

- Every function has one argument and one result.
- Any type can be passed/returned !!!
- Tuples are used to pass/return several arguments

```
- val a = (1.5, 6.8);
val a = (1.5, 6.8) : real * real
- fun lengthvec (x:real,y:real) = sqrt(x*x + y*y);
val lengthvec = fn : real * real -> real
- lengthvec a;
val it = 6.963476143 : real
- fun negvec (x:real,y:real) = (~x, ~y);
val negvec = fn : real * real -> real * real
- negvec (1.0, 1.0);
val it = (~1.0, ~1.0) : real * real
```

ML Introduction.21

Functions as Values

- Anonymous functions with **fn** notation

```
- fn x => x*x;
val it = fn : int -> int
- it(3);
val it = 9 : int
```

- The following declarations are identical
  - **fun** sq x = x*x;
  - **val** sq = **fn** x => x*x

ML Introduction.22

Notes

Functions as Parameters

- The definition of sigma:

$$\sum_{i=x}^{y} f(i) = \begin{cases} f(x) + \sum_{i=x+1}^{y} f(i) & \text{if } x \le y \\ 0 & \text{otherwise} \end{cases}$$

- Functions can be given as parameters to other functions

```
- fun Sigma(f,x,y) =
=   if x<=y then f(x) + Sigma(f,x+1,y)
=           else 0;
val Sigma =
  fn : (int -> int) * int * int -> int
- Sigma(sq,1,3);
val it = 14 : int
- Sigma(fn x => x*x,1,3);
val it = 14 : int
```

ML Introduction.23

Function as Return Value

- Functions can also be ***returned*** from other functions

```
- fun inttwice(f:(int->int)) =
      fn x => f(f(x));
val inttwice = fn : (int -> int) -> int -> int
```

- The arrow associates to the right so the last line is equivalent to

```
val inttwice = fn : (int -> int) -> (int -> int)
```

- Example

```
- inttwice(fn x => x*x);
val it = fn : int -> int
- it(3);
val it = 81 : int
```

ML Introduction.24

Notes

Type Inference

- ML deduces the types in expressions
- Type checking the function:

```
fun facti (n,p) =
  if n=0 then p else facti(n-1,n*p);
```

  - constants **`0`** and **`1`** have type **`int`**
  - therefore **`n=0`** and **`n-1`** involve integers
  - so **`n`** has type **`int`**
  - **`n*p`** must be integer multiplication, so **`p`** has type **`int`**
  - **`facti`** returns type **`int`**, and its argument type is **`int*int`**

ML Introduction.25

Type Constraints

- Certain functions are overloaded, e.g., abs,+,-,~,*,<.
- Type of an overloaded function is determined from context, or is set to ***int*** by default.
- Types can be stated explicitly.
- Examples:

```
- fun min(x,y) = if x < y then x else y;
val min = fn : int * int -> int
- fun min(x:real,y) = if x < y then x else y;
val min = fn : real * real -> real
- fun min(x:string,y) = if x < y then x else y;
val min = fn : string * string -> string
- fun min(x,y):real = if x < y then x else y;
val min = fn : real * real -> real
- fun min(x,y) = if x < y then x:real else y;
val min = fn : real * real -> real
```

ML Introduction.26

Notes

Polymorphic type checking

- Weakly typed languages (e.g.. Lisp)
  - give freedom
- Strongly typed languages (e.g. Pascal)
  - give security by restricting the freedom to make mistakes
- Polymorphic type checking in ML
  - security of strong type checking
  - great flexibility (like weak type checking)
  - most type information is deduced automatically
  - an object is polymorphic if it can be regarded as having any kind of type

ML Introduction.27

Polymorphic function definitions

- If type inference leaves some types completely unconstrained then the definition is polymorphic
- A polymorphic type contains a *type variable*, e.g. *'a*
- Example:

```
- fun pairself x = (x,x);
val pairself = fn : 'a -> 'a * 'a
- pairself 4.0;
val it = (4.0,4.0) : real * real
- pairself “NN”;
val it = (“NN”,”NN”) : string * string
- pairself (1.0,3);
val it =((1.0,3),(1.0,3)):(real*int)*(real*int)
- fun pair (x,y) = (y,x);
val pair = fn: (‘a * ’b) -> (‘b * ’a)
```

ML Introduction.28

Functions as Values

The Polymorphic Case

```
- fun twice f = fn x => f(f(x));
val twice = fn : ('a -> 'a) -> 'a -> 'a
- fun ident x = x;
val ident = fn : 'a -> 'a
- twice (fn x => x*x);
val it = fn : int -> int
- it(2);
val it = 16 : int
```

ML Introduction.29

Functions as Values

The Polymorphic Case (cont.)

- Sometimes ML gives us hard time when we give polymorphic value to polymorphic function. For example:

```
- twice ident;
stdIn:… Warning: type vars not generalized
because of value restriction are
instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 -> ?.X1
```

- The reason for this is outside the scope of this course. You usually may ignore it. Or, if needed, workaround the problem:

```
- fn x => (twice ident)(x);
val it = fn : 'a -> 'a
```

ML Introduction.30

Functional vs. Imperative

- Imperative - using commands to change the state.
- Functional - stateless. Using expressions recursively to calculate the result.
- Example: Euclid's Algorithm for the Greatest Common Divisor (GCD) of two natural numbers:

$$\gcd(m,n) = \begin{cases} n & m = 0 \\ \gcd(n \bmod m, m) & m > 0 \end{cases}$$

How would a GCD program would look like in functional vs. imperative language?

ML Introduction.31

GCD - Pascal vs. ML

- An imperative Pascal Program:

```
function gcd(m,n: integer): integer;
var prevm: integer;
begin
  while m<>0 do begin
       prevm := m; m := n mod m; n := prevm
  end;
  gcd := n
end;
```

- A functional program in Standard ML:

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

ML Introduction.32