ML

- Designed for Theorem Proving. (Edinburgh LCF) (1977)

- No way to defeat the type system.

- Module structure available.

- Allowed to raise and trap exceptions.
The ML family

- Sugared $\lambda$-calculus

- *call-by-value* semantics, so that arguments are evaluated before being passed to a function, unlike LISP’s lazy evaluation.

- Standard ML of NJ 1989

- Caml 1984
Introduction to Caml

You can obtain your own copy of Camllight from http://pauillac.inria.fr/caml. There is a PC version here, which I recommend to those of you who work at home without Linux.

We have Objective Cam installed. It is in /opt/local/products/ocaml-2.99

You will have to add /opt/local/products/ocaml-2.99/bin to your path.

Ocaml is not quite the same as Caml Light. But I have tested all the examples here on Ocaml as well as Caml light.
A simple Caml session

robinson:~/courses/ppl/code/caml$ ocaml
  Objective Caml version 2.99 (99/12/08)

# 1 + 2;;
- : int = 3
# (function x -> x + 1) (2 + 3);;
- : int = 6
# let successor = function x -> x + 1;;
val successor : int -> int = <fun>
# let square = function x -> x*x;;
val square : int -> int = <fun>
# square(successor 3);;
- : int = 16
# let addition x = function y -> x + y;;
val addition : int -> int -> int = <fun>
# addition 3 4;;
- : int = 7
Numbers

Integers: with $+,-,\times,\div$, and \texttt{mod}

Floats: with $+,-,\times,\div$. and \texttt{sqrt, exp, log, sin, cos, ...}
Newton’s Method

# let deriv f x dx =
  (f(x +. dx) -. f(x)) /. dx;;
val deriv :
  (float -> float) -> float ->
    float -> float = <fun>
# let abs x = if x > 0.0 then x else -.x;;
val abs : float -> float = <fun>
# let newton f epsilon =
  let rec until p change x =
    if p(x) then x
    else until p change (change(x)) in
  let satisfied y = abs( f y) < epsilon in
  let improve y =
    y -. (f(y) /. (deriv f y epsilon))
in until satisfied improve;;
val newton :
  (float -> float) -> float
    -> float -> float = <fun>
The session continued

```ocaml
# let square_root x epsilon =  
  newton (function y -> y *. y -. x) epsilon x ;;
val square_root : float -> float -> float = <fun>
# let cubic_root x epsilon =  
  newton (function y -> y *. y *. y -. x) epsilon x;
val cubic_root : float -> float -> float = <fun>

# square_root 2.0 1e-5 ;;
- : float = 1.41421569553

# cubic_root 3.0 1e-6 ;;
- : float = 1.44224963482

# 2.0 *. (newton cos 1e-5 1.5) ;;
- : float = 3.14159265359
```
Booleans and Equality

# true;;
- : bool = true
# false;;
- : bool = false

Functions with type bool are called *predicates*. There are <, >, <=, >=. (Dotted versions in Caml Light only).

# false = (1>2);;
- : bool = true
# (1,2) = (1,2);;
- : bool = true
# (2,1) > (1,2);;
- : bool = true
Conditionals

\[
\text{if } e_{test} \text{ then } e_1 \text{ else } e_2
\]

where \( e_{test} \) is a predicate and \( e_1 \) and \( e_2 \) are expressions of the same type.

# let negate a = if a then false else true ;;
val negate : bool -> bool = <fun>
# negate (1=2);;
- : bool = true
Logical operators.

or and & and not

or and & are not functions, since they evaluate their second argument only if it is needed.

```plaintext
# true or false;;
- : bool = true
# (1 < 2) & (2>1);;
- : bool = true
# not (1>2);;
- : bool = true
```

Note that not is treated a little differently. “not f x” is recognized as “not (f x)”, while normally “f g x” is “(f g) x”
Strings and Characters

- Strings between " double quotes.

- Single char constants between ’ single quotes.

- ^ is concatenation

```
# "Hello" ^ " World!";
- : string = "Hello World!"
# 'a';
- : char = 'a'
# '\065';
- : char = 'A'
```
Tuples

, is the value constructor for tuples.

# 1,2;;
- : int * int = 1, 2
# 1,2,3;;
- : int * int * int = 1, 2, 3
# let p = (1+2, (1<2));;
val p : int * bool = 3, true
Extracting pair components

# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
# fst (1,2);;
- : int = 1
# snd(1,2);;
- : int = 2

Note the polymorphic types 'a, 'b.
Pattern Matching 1

A *pattern* indicates the *shape* of a value.

# let f = function true -> false;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
false
val f : bool -> bool = <fun>
# f (1<2);;
- : bool = false
# f(2<1);;
Uncaught exception: Match_failure("", 8, 30)
Pattern Matching 2

# let imply = function (true,x) -> x  
  | (false, _) -> true;;
val imply : bool * bool -> bool = <fun>
# imply ((1>2), (4>5));;
- : bool = true

Note the use of the “don’t care” argument _.
Functions

# let compose f g = function x -> f (g (x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b =
# let succ x = x + 1;;
val succ : int -> int = <fun>
# (compose succ succ) 4 ;;
- : int = 6

# let add = function x -> (function y -> x + y);;
val add : int -> int -> int = <fun>
# let curry f = function x -> (function y -> f(x,y));
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f = function (x,y) -> f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
# add 3 4;;
- : int = 7
# plus (3,4);;
- : int = 7
# (curry plus) 3 4;;
- : int = 7
Lists 1

• [] is the empty list

• :: is the non-empty list constructor (cons).

• \([e_1; \ldots; e_n]\) is equivalent to \(e_1 :: (e_2 :: \ldots (e_n :: []) \ldots)\)

# 1::2::[];;
- : int list = [1; 2]
# [3;4;5];;
- : int list = [3; 4; 5]
# let x = 2 in [1;x+1;x+2];;
- : int list = [1; 3; 4]
Lists 2: Extracting arguments from Lists

# let is_null = function [] -> true | _ -> false;;
val is_null : 'a list -> bool = <fun>
# let head = function x:::_ -> x
     | _ -> raise (Failure "head");;
val head : 'a list -> 'a = <fun>
# let tail = function _:::t -> t
     | _ -> raise (Failure "tail");;
val tail : 'a list -> 'a list = <fun>
# tail [1;2;3;4;5];;
- : int list = [2; 3; 4; 5]
# tail [];;
Uncaught exception: Failure("tail")
Functions over lists

Old friends:

```ocaml
# let rec length = function [] -> 0
    | _ :: t -> 1 + length(t);;
val length : 'a list -> int = <fun>
# length [true;false];;
- : int = 2

# let rec append =
    function [], l2 -> l2
    | (x :: l1), l2 -> x :: (append (l1, l2));;
val append : 'a list * 'a list -> 'a list = <fun>
# append ([1;2], [3;4]);;
- : int list = [1; 2; 3; 4]
# [1;2] @ [3;4];;
- : int list = [1; 2; 3; 4]
```
Functions over lists

map

# let rec map f =
  function [] -> []
    | h::t -> (f h) :: (map f t);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (function x -> x + 1) [1;2;3;4;5;6;7];;
- : int list = [2; 3; 4; 5; 6; 7; 8]

# let rec it_list f a =
  function [] -> a
    | h::t -> it_list f (f a h) t;;
val it_list : ('a -> 'b -> 'a) ->
    'a -> 'b list -> 'a = <fun>
# it_list add 0 [1;2;3;4;5];;
- : int = 15

Recall the curried version of add

# let add = function x ->
  (function y -> x + y);;
val add : int -> int -> int = <fun>
User defined types

```haskell
# type person = { 
    name: string;
    age: int;
    job: string;
    city: string
 }

type person = { name: string; age: int;
    job: string; city: string } 

# let jean = { job="student"; city="Paris";
    name = "Jean"; age=23};;

val jean : person = {name="Jean"; age=23;
    job="student"; city="Paris"}

# let age_of = function
    {age=n; name=_; job=_;city=_} -> n;;

val age_of : person -> int = <fun>

# age_of jean;;
- : in t= 23

# jean.job;;
- : string = "student"
```
Parametrized types

Sum Types

# type ('a, 'b) pair = {
    fst: 'a;
    snd: 'b
};;
type ('a, 'b) pair = { fst: 'a; snd: 'b }
# let first x = x.fst and second x = x snd;;
val first : ('a, 'b) pair -> 'a = <fun>
val second : ('a, 'b) pair -> 'b = <fun>
# let p = { snd = true; fst=1+2};;
val p : (int, bool) pair = {fst=3; snd=true}
Sum Types

type identification =
  Name of string
  | SS of int * int

# let id1 = Name "jean";;
val id1 : identification = Name "jean"

# let id2 = SS(16342, 8668);;
val id2 : identification = SS (16342, 8668)
I/O

I/O is done on channels. These are abstract types. We have std_in, std_out and std_err

#print_char;;
- : char -> unit = <fun>
#print_string;;
- : string -> unit = <fun>
#print_int;;
- : int -> unit = <fun>
#print_float;;
- : float -> unit = <fun>
#flush;;
- : out_channel -> unit = <fun>
#print_newline;;
- : unit -> unit = <fun>
More I/O

#read_line;;
- : unit -> string = <fun>
#read_int;;
- : unit -> int = <fun>
#read_float;;
- : unit -> float = <fun>
#output_char;;
- : out_channel -> char -> unit = <fun>
#output_string;;
- : out_channel -> string -> unit = <fun>
#input_char;;
- : in_channel -> char = <fun>
#input_line;;
- : in_channel -> string = <fun>
Exceptions

Exceptions are defined with

\[
\text{exception } \langle \text{exception name } \rangle \ [\text{of type}]
\]

Eg.

exception Found of int;;

Here is an example of their use.

(* Don’t program this way *)

#exception Found of int;;
Exception Found defined.
#let find_index p L =
  let rec find n =
    function [] -> raise (Failure "not found")
    | x::L -> if p(x) then raise (Found n)
      else find (n+1) L
  in
  try find 1 L with Found n -> n;;

>let find_index p L =
>
Warning: the variable L starts with an upper case letter in this pattern.
>
>  | x::L -> if p(x) then raise (Found n)
>
Warning: the variable L starts with an upper case letter in this pattern.
find_index : ('a -> bool) -> 'a list -> int = <fun>

#find_index (function n -> (n mod 2) = 0) [1;2;3];;
- : int = 2