FCPL

Laboratory

Faculty of Automations and Computers
Department of Computers

Laboratory 10

Subjects

ML

ML (Meta-Language) is a set of advanced functional programming languages. The language combines the properties of LISP and Algol and is the first language to include polymorphic typing. ML was designed at the Univerisity of Edimburgh in 1973. Just like LISP, ML has several dialects (Standard ML, Lazy ML, CAML, etc.)
The CAML (original Categorical Abstract Machine Language) dialect is a french version of ML which supports, among other things, object oriented programming (OCAML), as well as iterative programming.
The original ML was a language build for writing software which manipulates other software - like compilators and interpreters, but along the way became a standalone language, most often used in software validation and automatical demonstration of mathematical theorems. It is also used in WEB development, protocol comunication or distributive calculations.

CAML Properties

- Functional language
  Functions are treated like any other data type, they can be used as a parameter or a return value for other functions. Functional programming allows coding without side effects.
- Well defined data types
  The language offers the programmer a set of data types and operators needed for using them. You can also define new data types.
- Strongly typed language
  Declaring a variable's type is not necessary, it will be automatically determined. The function parameters will be checked during compilation (not only at runtime).
- Polymorphic types
  You can declare functions without specifying the parameter's type and execute them with any type.
- Pattern matching
  ML allows writing rule based software (see laboratory 11).
- Automatic garbage collector
  Memory management is done automatically by the language.
- Security
  The code is checked for errors at compile time.
- Imperative programming properties
  Mechanisms for manipulating tables and variables that change their content.
- Error handling
  Exception handling mechanism.

- Object oriented programming
  CAML has OOP features.

Starting CAML

Under **Linux**, CAML can run in emacs as an inferior process, similary to the xlispstat interpreter. In order to run the interpreter, you must run the caml inferior process with `M-x run-caml`. (Requires accepting the ocamil interpreter by pressing Enter).
In the ML workspace you must set the mode to *caml*. You can do this by running this: `M-x caml-mode`.
Under **Windows** CAML does not have an interface in emacs. CAML runs as an interpreter that reads commands line by line. You can start caml by selecting the Objective Caml icon on the desktop. At startup, the ML interpreter appears in interactive mode: a read, evaluate, display cycle, where the user types an expression which will be evaluated and the result displayed. # is the prompter after which ML functions can be written. The interpreter will display the result on a new line. In the laboratory, Caml is only installed on the Taurus, Cancer, Aquarius, Pisces computers, due to disk space limitations.

Simple ML Expression

```
# 2+3;;
- : int = 5
```

The examples are given as command-response. The lines starting with `#` must be typed (without `#`) and evaluated; the result (received after pressing `M-C-X` or `Enter`) is shown in a lighter colour.

We can write an expression on multiple lines and end it with `;;`, which determines the evaluation and result displaying.

Primitives

Integers and real numbers

CAML has two number data types: integers and real numbers. The two types are different and have different operation sets:
Integers:

`+`	sum
-	subtraction
`*`	multiplication
`/`	integer division
`mod`	the rest of the division

Real numbers:

`+.`	sum
`-.`	subtraction
`*.`	multiplication
`/.`	integer division
`**`	exponentiation

Operators

```
# 2 + 3;;
- : int = 5
# 4.0 +. 5.5;;
- : float = 9.5
# 4.0 +. 5;;
```

```
Characters 7-8:
4.0 +. 5;;
       ^
This expression has type int but is here used with type float
```

Problem 1

Evaluate the examples above.

Character types and strings

Characters are ASCII codes between `0` and `255`. They are typed between apostrophes.
Character string can have a length of maximum $2^{24}-6$.
The ^ merges two strings.

Character types and strings

```
# 'a';;
- : char = 'a'
# "the result" ^ " is " ^ "a string";;
- : string = "the result is a string"
```

Boolean type

The boolean type can hold the value of `true` or `false`.
Allowed operations:

`&& or &`	logic AND
`|| or or`	logic OR
`not`	login NOT

Logical operators

```
# true && false;;
- : bool = false
# false or not (2=3);;
- : bool = true
```

Problem 2.

Evaluate the examples above.

Logical operators only evaluate arguments until the result can be determined. The rest of the arguments are not evaluated.

Comparison operators

The =, <, <=, >, >=, <> operators are polymorphic, they can compare numbers, characters or strings.

Comparison operators

```
# 5 < 6;;
- : bool = true
# "beta" > "alfa";;
- : bool = true
```

Lists

Lists are infinite data structures. The lists are either void or consisting of the head of the list and the rest of the elements. All elements of the list must be of the same type.

Lists

```
# [];;
- : 'a list = []
# [1;2;3];;
- : int list = [1; 2; 3]
```

Observations: `int list` shows that the result is a list of integers and `ï¿½a list` shows that the result is a list of unspecified data type (meaning that it could be any type).

The `::` operator adds a new element at the beginning of the list.
The `@` operator merges two lists.

List operators

```
# 1::2::3::[];;
- : int list = [1; 2; 3]
# ['a';'b'] @ ['c';'d';'e'];;
- : char list = ['a'; 'b'; 'c'; 'd'; 'e']
```

Problem 3.

Evaluate the examples above.

Other operations for the list type are available in the List library. The `hd` and `tl` functions return the first element of the list, respectively the tail of the list.

Head and tail

```
# List.hd [1;2;3;4];;
- : int = 1
# List.tl [1;2;3;4];;
- : int list = [2; 3; 4]
```

N-tuple

Aggregating the values of different data types can generate n-tuples. N-tuples are typed as an enumeration of values, separated by commas inside round brackets.

N-tuple

```
# (2,'D',3,"theee") ;;
- : int * char * int * string = (2, 'D', 3, "three")
```

Conditional structures

Like other languages, ML has conditional structures. Unlike iterative languages, the evaluation returns a value.
Syntax:

```
    if expr1 then expr2 else expr3
```

The expression will be evaluated to `expr2` if `expr1` is true and to `expr3` if `expr1` is false.

IF ... THEN ... ELSE

```
# if 2=3 then 2 else 3;;
- : int = 3
```

Global variables

Using the let primitive, you can link the value of an expression to a variable.

LET

```
# let a = 5 * 2;;
val a : int = 10
# a ;;
- : int = 10
```

Parallel declarations can be done using the following syntax:

```
let var1 = expr1
and var2 = expr2
...
and varn = exprn ;;
```

Declared variables will not be able to access the rest of the variables until the end of `let`.

Parallel LET

```
# let a = 1;;
val a : int = 1
# let a = 2
and b = a + 1;;
val a : int = 2
val b : int = 2
# a + b ;;
- : int = 4
```

You can also declare variables sequentially. They will be able to access all previously declared variables. For this, we chain multiple instructions. The declaration sequence will end with `;;`.

Sequential LET

```
# let z = 1
let u = z+2;;
val z : int = 1
val u : int = 3
```

Local declarations

Local declarations are used in order to limit the domain of existence of a variable.
Local declarations are done by using the `let` primitive with the before mentioned syntax, followed by `in` and the expression in which we want it to be used in.

Local declarations

```
# let x = 1 and y = 2 in x+y;;
- : int = 3
```

Functions

A function is defined by using the following syntax:

```
    function p -> expr
```

This definition is similar to the lambda declarations in LISP.
An expression defined with `function` results in a function which can be called.

Function definition

```
# function x -> x + 1 ;;
- : int -> int = <fun>
# (function x -> x + 1) 2;;
- : int = 3
```

The expression can be another function.

Function definition

The following examples will show how you can declare functions with multiple parameters.

```
# function x -> (function y -> x + y + 1) ;;
- : int -> int -> int = <fun>
# function x -> function y -> x + y + 1 ;; (* equivalent
declaration*)
- : int -> int -> int = <fun>
# (function x -> function y -> x + y + 1) 2;; (* returns y ->
2 + y + 1 *)
- : int -> int = <fun>
# (function x -> function y -> x + y + 1) 2 5;;
- : int = 8
```

Functions can also have, as parameters, n-tuples.

Function definition

```
# (function (x,y) -> x + y+ 1) (2,5);;
- : int = 8
```

Observation: this declaration is fundamentally different from the previous one. Here, the function expects a single parameter, while the previous one expects two.

Alternatively, in CAML, we can use the `fun` keyword.
`fun p1 p2 ...pn -> expr`
is equivalent to
`function p1 ->function p2 ->... -> function pn -> expr`

Function definition

```
# (fun x y -> x + y + 1) 2 5;;
- : int = 8
```

In the declarations above, the functions had no name assigned to them. A `function` expression return a function which can be assigned a name by using `let`:

Function definition

```
# let addinc = function x -> function y -> x + y + 1;;
val addinc : int -> int -> int = <fun>
# addinc 2 5;;
- : int = 8
```

Compact typing is also allowed:
`let name p1 ... pn = expr`
is equivalent to
`let name = function p1 -> -> function pn -> expr`

Function definition

```
# let addinc x y = x + y + 1;;
val addinc : int -> int -> int = <fun>
```

If we want to define recursive functions, then the name must be assigned to them with `let rec`.

Factorial

```
# let rec fact=function n ->
    if n=0 then 1 else n*fact (n-1);;
val fact : int -> int = <fun>
# fact 3;;
6
```

Problem 4.

Implement Fibonacciâ€™s function in ML.

Problem 5.

Implement the Ackerman function.

```
ack(x,y)= ack(x-1, ack(x,y-1)), dacă x>0, y>0
          ack (x-1,1), dacă x>0, y=0
          y+1, dacă x=0
```

Problem 6.

Write the `geninterval`, function which has two parameters: `s` and `e` and return a list of the integers between `s` and `e`.

```
# geninterval 3 8;;
- : int list = [3; 4; 5; 6; 7; 8]
```

Problem 7.

Define the functions `my_not`, `my_and` and `my_or` without using the `not`, `and` or `or` operators.

```
# my_and (3=3) (4=4);;
- : bool = true
```

Problem 8.

Write the function `digits` that converst an integer in a list of it digits

```
# digits 123;;
- : int list = [1; 2; 3]
```

Problem 9.

Write the function `maxim` which returns the maximum value in a list. You can use the `max` function, which returns the maximum of its two parameters.

```
# maxim [2;3;1;7;5];;
- : int = 7
```

Problem 10.

Write the function `GCF`, greatest common factor.

Installing OCAML

You can get the latest OCAML version here. The 3.07 version, which is installed in the laboratory, can be found here for Linux RH7.3 and here for Windows.

For Windows

Download the OCAML executable from the INTRA website or from the local server. Run the executable and follow the installation steps. Under Windows, the interface with emacs is not available!

For Linux

Download the precompiled archive or the sources from INRA. The RPM precompiled version for Linux Redhat 7.3 can be found, locally, here. Install the archive. For using the emacs interface, you have to add the following lines in the .emacs file found in the home directory:

```
(setq auto-mode-alist (cons '("\\.ml[iylp]?" . caml-mode)
               auto-mode-alist))
(autoload 'caml-mode "caml"
               "Major mode for editing Caml code." t)
(autoload 'run-caml "inf-caml"
               "Run an inferior Caml process." t)
(autoload 'camldebug "camldebug"
               "Run the Caml debugger." t)
```

Here you can find the .emacs file containing the previously mentioned lines, which are necessary for xlispstat and OCAML.

Problems

Problem 1. Numeric types
Problem 2. Logical operations
Problem 3. List operations
Problem 4. Fibonacciâ€™s function
Problem 5. Ackerman's function
Problem 6. List generator
Problem 7. Logical operators
Problem 8. Big numbers
Problem 9. Maximum in a list
Problem 10. Greatest common factor