Cflp

Laborator

Facultatea de Automatica and Calculatoare
Departamentul de Calculatoare



Lucrarea 11

# Pattern matching

Subjects

Pattern matching is a flexible selection mechanism, which, depending on an expression's value, selects the result's value. The mechanism is similar to the `case` instruction in Pascal or the `switch` instruction in C, but with a more powerful range of facilities.

## MATCH

MATCH

Syntax:

```
match expr with
      | p1 -> expr1
        ...
      | pn -> exprn
```

FUNCTION

The instruction `match` evaluates the expression `expr` and compares it with the options `p1...pn`, one by one. If one matches `pi` , then the result of `match` will be `expri`.

TYPE

**mynot**

```
# let mynot p = match p with true -> false | false -> true;;
val mynot : bool -> bool = <fun>
```

The article

It is adivsable that the options cover the entire value domain of the tested expression. CAML checks this and gives a warning in case the defintion of `match` is incomplete.

**Completeness of the test cases**

Variable types

```
# let test p = match p with 0 -> true | 1 -> false;;
Characters 13-48:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
let test p = match p with 0 -> true | 1 -> false;;
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
val test : int -> bool = <fun>
```

Problems

**Exclusive OR 1**

```
# let xor p = match p with
      (true,true)   -> false
    | (true,false)  -> true
    | (false,true)  -> true
    | (false,false) -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, false);;
- : bool = true
```

Expressions may also contain variables. A pattern containing each variable only once is called a linear pattern.

### Exclusive OR 2

This is allowed:

```
# let xor p = match p with
      (true, x)  -> not x
    | (false, x) -> x;;
val xor : bool * bool -> bool = <fun>
```

But this is not allowed:

```
# let xor p = match p with
      (x, x) -> false
    | (false, true) -> true
    | (true, false) -> true;;
Characters 33-34:
(x, x) -> false
^
This variable is bound several times in this matching
```

## Compacting cases

The _ symbol is used to define the *default*, its value matches any expression.

### Exclussive OR 3

```
# let xor p = match p with
      (false, true) -> true
    | (true, false) -> true
    |  _ -> false;;
val xor : bool * bool -> bool = <fun>
```

By using the | (pipe) symbol, you can combine multiple patterns for which the result is the same.

### Test if an integer is a digit

```
# let digit n = match n with
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 -> true
 | _ -> false;;
val digit : int -> bool = <fun>
```

# Pattern matching function parameters

Using pattern matching in multiple case functions is essential in ML.
A more precise syntax of the `function`, keyword (presented in the previous laboratory) is:

```
function | p1 -> expr1
         | p2 -> expr2
         ...
         | pn -> exprn
```

This syntax is equivalent to

```
function expr -> match expr with
           p1 -> expr1
         | p2 -> expr2
         | ...
         | pn -> exprn
```

### Testing for 0 and 1

```
# let test01 = function | 0 -> true
      | 1 -> true
```

```
      |  _ -> false;;
val test01 : int -> bool = <fun>
# test01 1;;
- : bool = true
# test01 2;;
- : bool = false
```

### List pattern matching

Manipulating lists can be easily done by implementing pattern matching. For this, the `::` and `@` operators are very useful.

#### Computing the lenght of a list

```
# let rec length p = match p with
      [] -> 0
    | head::tail -> (1+length tail);;
```

#### Problem 1.

Define the `myand` and `myor` functions using pattern matching.

#### Problem 2.

Define the `headoflist` and the `tailoflist` functions using pattern matching. Do not use `List.hd` or `List.tl`.

#### Problem 3.

Define the `reverse` function which reverses the items in list. Do not use `List.hd`, `List.tl` or the functions defined at Problem 2.

#### Problem 4.

Define the `rotate_left` and the `rotate_right` functions, which rotate a list to the left or right. Do not use `List.hd` , `List.tl` or the functions from Problem 2.

#### Problem 5.

Define the `maxoflist` function which determines the maximum value found in a list with any type of values, using pattern matching. Do not use `List.hd` , `List.tl` or the functions from Problem 2.

#### Problem 6.

Define the `apply f i [l1::l2::..ln]` function which receives a function `f`, an initial value `i` and a list and returns: `f(l1, f(l2, ... f(ln1, f(ln,i))...))`. Do not use `List.hd` , `List.tl` or the functions from Problem 2.

```
# let u x y = x + y;;
# let f x y = 2 * x + y;;
# apply u 0 [1;2;3;4];;
- : int = 10
# apply f 1 [1;2;3;4];;
- : int = 21
```

# Data type declarations

You can declare new data types using the `type` keyword.

Syntax:

```
type  type1 = typedef1
  and tip2 = typedef2
  ...
  and typen = typedefn ;;
```

Where `typei` is equivalent to `typedefi`.

### Data type declarations

```
# type t1 = (int*int)
   and t2 = (int*char);;
type t1 = int * int
type t2 = int * char
```

You can also add parameters to the declaration:

```
type 'a tip = typedef ;;
type ('a1 ...'an) tip = typedef ;;
```

### Parameterized type

In the next example, the `pair` type is declared. The pair is formed by an integer and a data type specified by the user.

```
# type 'customtype pair = int * 'customtype;;
type 'a pair = int * 'a
```

Specifying a more generic type is also allowed.

### Specializarea tipurilor

```
# type pair_char = char pair;;
type pair_char = char pair
```

If the data type cannot be inferred, we must explicitly specify it.

### Specifying the type

```
# let variabila=(1,'a');;
val variabila : int * char = (1, 'a')
# let (variabila:pereche_char)=(1,'a');;
val variabila : pereche_char = (1, 'a')
```

## The article

N-tuples are not flexible enough. Just like in C or Pascal, ML offers the possibility of using records, each element of a record having it's own name. Declaring a record is done using the syntax:

```
type record = { field1 : type1; ...; fieldn : typen } ;;
```

### Rational numbers

```
# type ratnum = { num: float ; den : float};;
type ratnum = { num : float; den : float; }
```

Record's fields can be assigned in an arbitrary order.

```
{ field1 = expr1; ...; fieldn = exprn } ;;
```

### Rational number

```
# let numar = {den =2.;num = 3.};;
val numar : numarrat = {num = 3.; den = 2.}
# numar = {num = 1.+.2.; den = 2.};;
- : bool = true
```

Accessing the value of a field can be done using the regular dot notation.

An alternative is pattern matching using the syntax:
`{ namei = pi ; ...; namej = pj }`
Where `pi` are patterns usually formed out of variables. It is not necessary to enumerate all fields of the record.

### Incrementation of a rational number

```
# let increment p = {num = p.num +. p.den; den = p.den};;
val increment : numarrat -> numarrat = <fun>
# let increment p = match p with
   { num = n ; den = d} -> {num = n+.d; den = d};;
val increment : numarrat -> numarrat = <fun>
# increment {num = 4.; den = 2.};;
- : numarrat = {num = 6.; den = 2.}
```

## Types with choices

Types with variants are similar with the `union` type from C. Depending on a selector, the type may have different fields.

```
type nume = ...
     | Constructori ...
     | Constructorj of tipj ...
     | Constructork of tipk * ...* tipl ...;;
```

`Constructorx` is called constructor and it is a special identifier. **The constructor shall always start with an uppercase letter.**

### Declaring types with variants

```
# type calificativ = Admis | Respins;;
type calificativ = Admis | Respins
# type nota = Patru | Cinci | Sase | Sapte | Opt | Noua |
Zece;;
type nota = Patru | Cinci | Sase | Sapte | Opt | Noua | Zece
# type examen = Examen of nota*prezentare
          | Colocviu of calificativ;;
# type prezentare = int;;
type prezentare = int
# type examen = Examen of nota * prezentare
          | Colocviu of calificativ
type examen = Examen of nota * prezentare | Colocviu of
calificativ
```

Initializing a variable is done using a constructor and a value.

### Instantiating types with variants

```
# let practica = Colocviu Admis;;
val practica : examen = Colocviu Admis
# let cflp1 = Examen(Zece,1);;
val cflp1 : examen = Examen (Zece, 1)
```

Processing variables with variants can be done using pattern matching.

### Converting type examen to string

```
# let string_of_calificativ = function
      Admis -> "admis."
    | Respins -> "respins.";;
val string_of_calificativ : calificativ -> string = <fun>
# let string_of_nota = function
      Patru -> "patru"
    | Cinci -> "cinci"
```

```
    | Sase  -> "sase"
    | Sapte -> "sapte"
    | Opt   -> "opt"
    | Noua  -> "noua"
    | Zece  -> "zece";;
val string_of_nota : nota -> string = <fun>
# let string_of_examen = function
      Colocviu calif -> "Colocviul cu calificativ " ^
      string_of_calificativ calif
    | Examen (n,p) -> "Prezentarea " ^
      string_of_int p ^ " cu nota " ^ string_of_nota n;;
val string_of_examen : examen -> string = <fun>
# string_of_examen cflp1;;
- : string = "Prezentarea 1 cu nota zece"
```

**Problem 7.**

Define type`nrcomplex` define functions for complex addition and multiplication.

# Probleme