

CS 162 Programming Languages

Lecture 3: OCaml Crash Course II

Yu Feng
Winter 2026

Put it together: a “filter” function

If arg matches
this pattern

then use this
body expr

```
# let rec filter f l =  
  match l with  
  | [] -> []  
  | (h::t)-> if f h then h::(filter f t)  
              else (filter f t);;  
  
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>  
  
# let list1 = [1;31;12;4;7;2;10];;  
# filter is5lt list1 ;;  
val it : int list = [31;12;7;10]
```

Put it together: a “quicksort” function

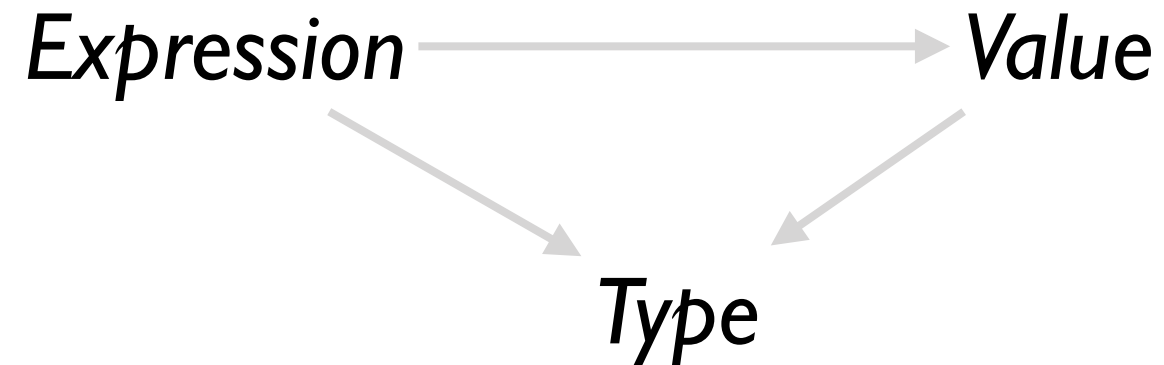
```
# let partition f l = (filter f l, filter (neg f) l);;  
val partition : ('a->bool)->'a list->'a list * 'a list = fn  
# let list1 = [1;31;12;4;7;2;10];  
# partition is5lt list1 ;  
val it : (int list * int list) = ([31;12;7;10],[1;4;2])
```

```
# let rec sort l =  
    match l with  
    [] -> []  
  | (h::t) ->  
    let (l,r) = partition ((<) h) t in  
    (sort l)@(h::(sort r)) ;;
```

Outline for today

- Data types
- Pattern matching

ML's holy grail



- Simple
- Variables
- Functions

Building datatypes

Three key ways to build complex types/values

- “Each-of” types:
Value of T contains value of T1 and a value of T2
- “One-of” types:
Value of T contains value of T1 or a value of T2
- “Recursive”
Value of T contains (sub)-value of same type T

One-of types

We've defined a "one-of" type
named attrib

Elements are one of:

- string
- int
- int*int*int
- float
- bool

```
type attrib =  
    Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of real  
| Alive of bool  
| Phone of int*int  
| Email of string;
```

Each-of types

We've defined a "Each-of" type (i.e., product type) named "DOB" attrib is the composition of three ints:

- `int*int*int`

```
type attrib =  
    Name of string  
| DOB of int*int*int
```


Test & Take whats in box?



Is it a ...
string?
or an int?
or ...

Check the TAG!

Whats in the box

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of real  
| Alive of bool  
| Phone of int*int  
| Email of string;
```

```
match e with  
| Name s -> printf "%s" s  
| Age i -> printf "%d" i  
| DOB(d,m,y) -> printf "%d/%d/%d" d m y  
| Address s -> printf "%s" s  
| Height h -> printf "%f" h  
| Alive b -> printf "%b" b  
| Phone(a,r) -> printf "(%d)-%d" a r
```

Pattern-match expression: check if e is of the form ...

- On match:
 - value in box bound to pattern variable
 - matching result expression is evaluated
- Simultaneously test and extract contents of box

Beware to handle all tags!

```
# match (Name "Bob") with  
| Age i -> Printf.printf "%d" i  
| Email s -> Printf.printf "%s" s ;;
```

Exception: Match Failure!!

None of the cases matched the tag (Name)
Causes nasty Run-Time Error!

Compiler to rescue!

```
# # let printAttrib a = match a with
| Name s -> Printf.printf "%s" s
| Age i -> Printf.printf "%d" i
| DOB (d,m,y) -> Printf.printf "%d / %d / %d" d m y
| Address addr -> Printf.printf "%s" addr
| Height h -> Printf.printf "%f" h
| Alive b -> Printf.printf "%b" b
| Email e -> Printf.printf "%s" e
;;
```

Warning P: this pattern-matching is not exhaustive. Here is an example of a value that is not matched: Phone (_, _)

Compile-time checks for:
missed cases: ML warns if you miss a case!

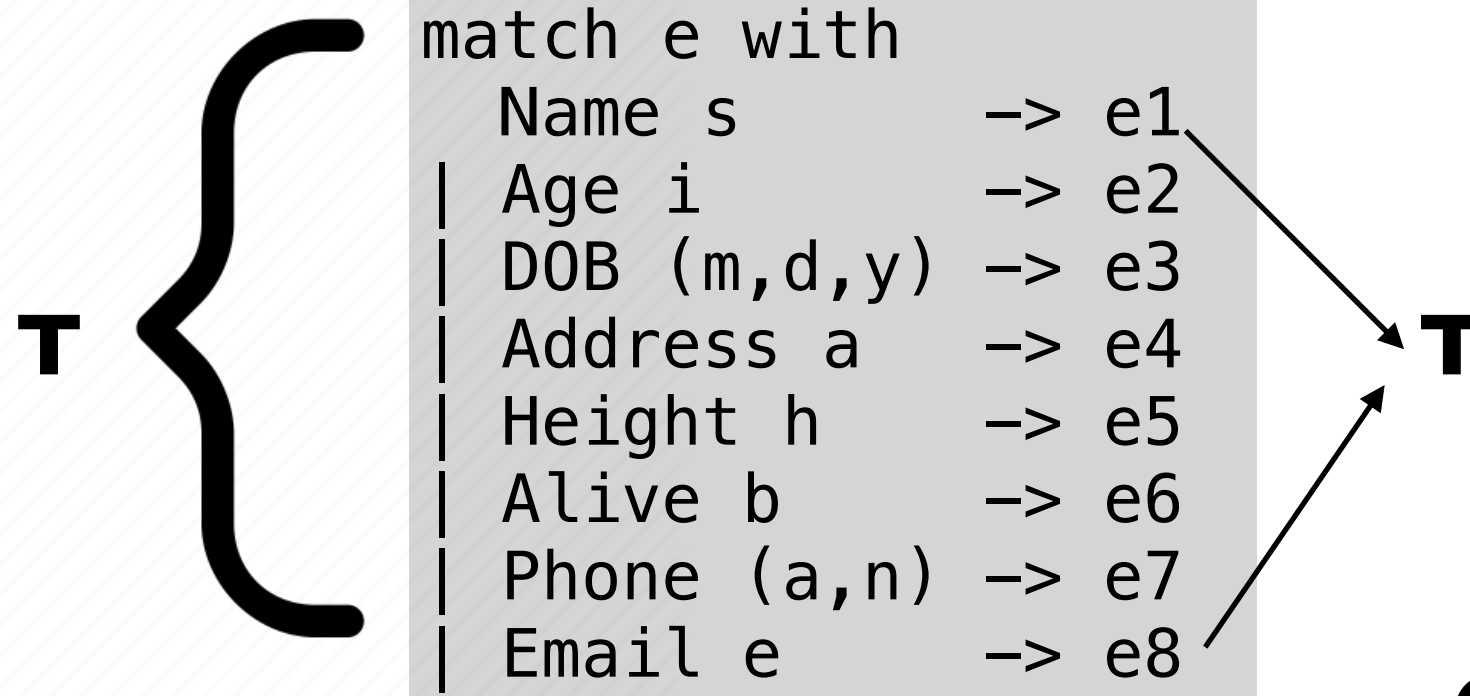
match-with is an Expression

```
match e with
  C1 x1 -> e1
| C2 x2 -> e2
| ...
| Cn xn -> en
```

Type Rule

- e_1, e_2, \dots, e_n must have same type T
- Type of whole expression is T

match-with is an Expression



Type Rule

- `e1, e2, ..., en` must have same type `T`
- Type of whole expression is `T`

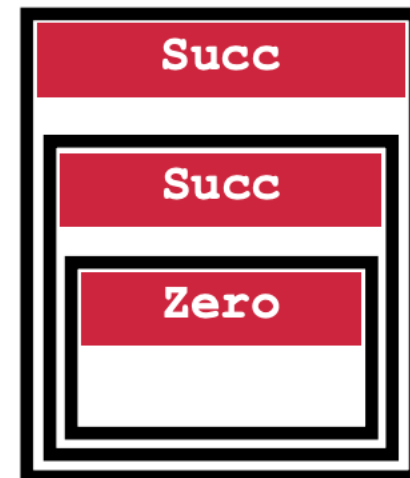


Recursive types

```
type nat = Zero | Succ of nat
```

What are values of nat ?
One nat contains another!

nat = recursive type



plus: nat*nat -> nat

Base pattern
Inductive pattern

```
type nat =  
| Zero  
| Succ of nat
```

Base pattern
Inductive pattern

```
let rec plus n m =  
match m with  
| Zero -> n  
| Succ m' -> Succ (plus n m')
```

Base expression

Inductive expression

List datatype

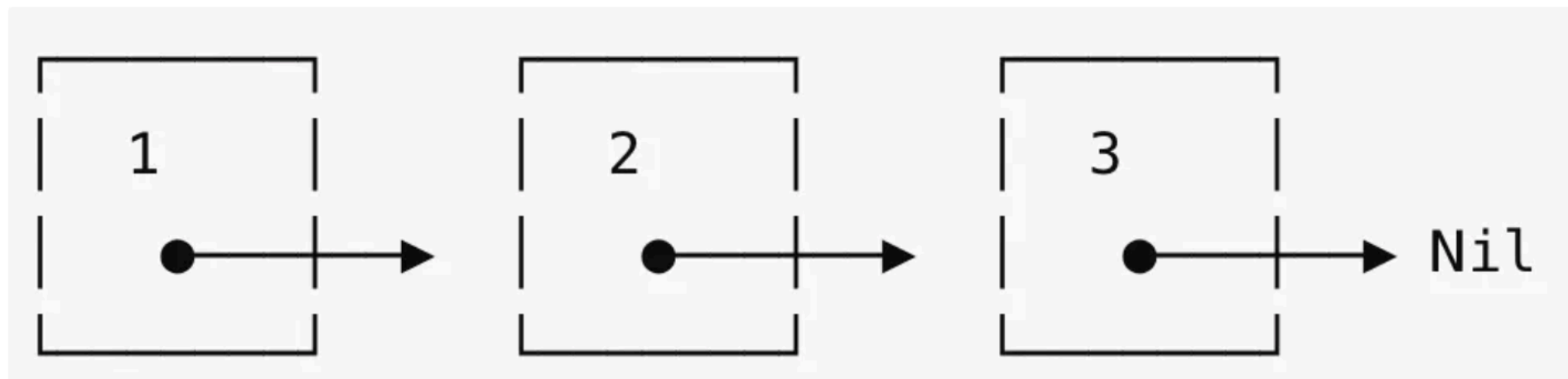
```
type int_list =  
  Nil  
| Cons of int * int_list
```

Lists are a derived type: built using elegant core!

1. Each-of
2. One-of
3. Recursive

:: is just a syntactic sugar for “Cons”
[] is a syntactic sugar for “Nil”

Cons (1, Cons (2, Cons (3, Nil)))



List function: length

```
let rec len l =  
  match l with  
    Base pattern | Nil -> 0 Base expression  
    Inductive pattern | Cons(h,t) -> 1 + (len t)  
                                Inductive expression
```

List function: list_max

```
let rec list_max l =  
  match l with  
    Base pattern | Nil -> 0 Base expression  
    Inductive pattern | Cons(h,t) -> max h (list_max t)  
                                         Inductive expression
```

```
let max x y = if x > y then x else y;;
```

Option types

```
type 'a option = Some of 'a | None
```

```
type 'a btree = {  
  value : 'a;  
  left  : 'a btree option;  
  right : 'a btree option;  
}
```

```
match tree with  
| None -> []  
| Some node -> ...
```

an option type is a built-in data type that represents a value that can either be present or absent. It's a way to safely handle cases where a value might be missing, rather than using something like null (which can lead to runtime errors).

TODOs by next lecture

- Come to the discussion session if you have questions
- Start to work on HW1