CS 162 Programming Languages

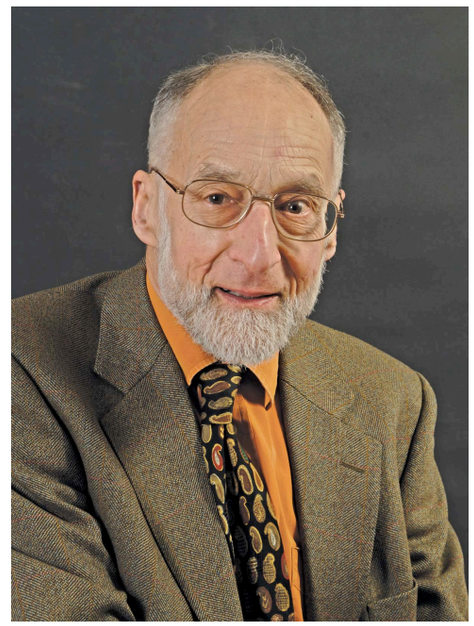
Lecture 2: OCaml Crash Course I

Yu Feng

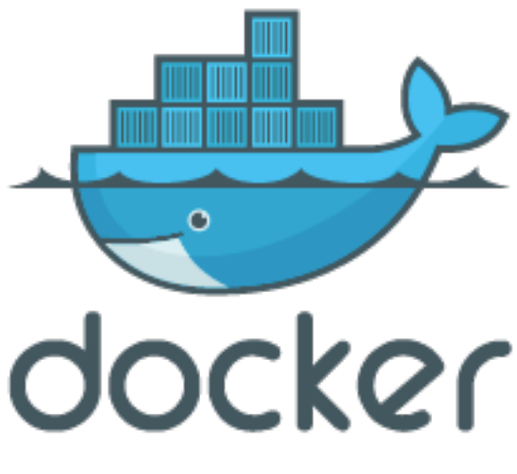
Winter 2025

History of ML

- ML = "Meta Language"
- Designed by Robin Milner @ Edinburgh
- Language to manipulate Theorems/Proofs
- Several dialects:
  - Standard" ML (of New Jersey)
  - French dialect with support for objects
  - State-of-the-art
  - Extensive library, tool, user support



Who are using OCaml



OCaml vs. C

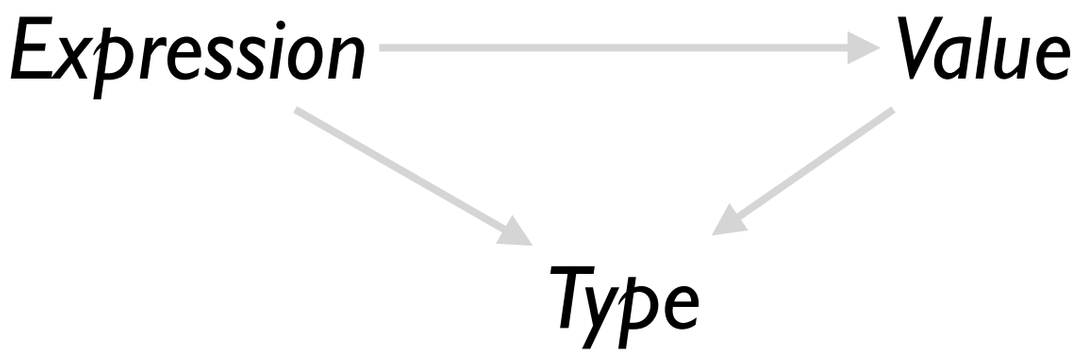
```
void sort(int arr[], int beg, int end){
  if (end > beg + 1){
    int piv = arr[beg];
    int l = beg + 1;
    int r = end;
    while (l != r-1){
       if(arr[l] <= piv)
          l++;
       else
          swap(&arr[l], &arr[r--]);
    }
    if(arr[l]<=piv && arr[r]<=piv)
       l=r+1;
    else if(arr[l]<=piv && arr[r]>piv)
       {l++; r--;}
    else if (arr[l]>piv && arr[r]<=piv)
       swap(&arr[l++], &arr[r--]);
    else
       r=l-1;
    swap(&arr[r--], &arr[beg]);
    sort(arr, beg, r);
    sort(arr, l, end);
  }
}
```

Quicksort in C

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

Quicksort in Ocaml

ML's holy grail



- Everything is an expression
- Everything has a value
- Everything has a type

Interacting with ML

"Read-Eval-Print" Loop

Repeat:
1. System reads expression e
2. System evaluates e to get value v
3. System prints value v and type t

What are these expressions, values and types ?

Basic types

```
# 2;;
```
2

```
# 2+3;;
```
Int **5**

```
# “hi”;;
```
“hi”

```
# “hi,”^”OCaml”;;
```
String **“hi, OCaml”**

```
# true;;
```
true

```
# 2>3;;
```
Bool **false**

Type errors

```
# “Hi,” ^ 2;;
# (2+3) || 9;;
```

Untypable expression is rejected

- No casting or coercing
- Fancy algorithm to catch errors
- ML’s single most powerful feature

Complex types: Lists

List operators:

- Cons (::): "cons" element to a list of same type
- append (@): only append two list of the same type
- Head (List.hd): return the head element of a nonempty list
- Tail (List.tl): return the tail of nonempty list

Syntax:

- Lists = semicolon

Semantics:

- Same type, unbounded number

Complex types: Tuples

```
# (9-3,“ab”^“cd”,(2+2,7>8));;
```

(6,”abcd”,(4,false))

int * string * (int * bool))

Syntax:

- Lists = comma

Semantics:

- Different type, fixed number

Variables and bindings

`let` `x = e`

"Bind the value of expression e to the variable x"

```
# let x = 2+2;;
val x : int = 4
```

Variables and bindings

Later declared expressions can use x

- Most recent "bound" value used for evaluation

```
# let x = 2+2;;
val x : int = 4
# let y = x * x * x;;
val y : int = 64
# let z = [x;y;x+y];;
val z : int list = [4;64;68]
```

Variables and bindings

Undeclared variables (i.e. without a value binding)
are not accepted !

```
# let p = a + 1;
Characters 8-9:
    let p = a + 1 ;;
^ Unbound value a
```

Local bindings

for expressions using "temporary" variables

```
# let
    tempVar = x + 2 * y
  in
    tempVar * tempVar ;;
```

- tempVar is bound only inside expr body from in … ;;
- Not visible ("in scope") outside

Complex types: functions

```
# fun x -> x+1;;
```

<fun>

int->int

Parameter (formal)

Body (expr)

```
# let inc = fun x -> x+1 ;
val inc : int -> int = fn
# inc 0;
val it : int = 1
# inc 10;
val it : int = 11
```

How to evaluate a function app:

- Evaluate the argument
- Bind formal to arg value
- Evaluate the "body expr"

Complex types: functions

```
# fun x -> fun y -> x<y;;
```



```
<fun>
```

```
'a->('a->bool)
```

Wow! A function can return a function

```
# let lt = fun x -> fun y -> x < y;;
val lt : 'a -> 'a -> bool = fn
# let is5Lt = lt 5;
val is5lt : int -> bool = fn;;
# is5lt 10;;
val it : bool = true;
# is5lt 2;;
val it : bool = false;
```

Complex types: functions

```
# fun f -> fun x -> not (f x);;
```



```
<fun>
```

```
('a->bool)->('a->bool)
```

A function can also take a function argument

```
# let neg = fun f -> fun x -> not (f x);
val lt : (a -> bool) -> a -> bool = fn
# let is5gte = neg is5lt;
val is5gte : int -> bool = fn
# is5gte 10;
val it : bool = false;
# is5gte 2;
val it : bool = true;
```

Pattern matching

A pattern matching is somewhat similar to switch statement but offers a lot more expressive power. It really boils down to matching an argument against an exact value, a predicate, or a type constructor.

```
type animal = Dog of string | Cat of string ;;
```

```
let say x =
    match x with
    | Dog x -> x ^ " says woof"
    | Cat x -> x ^ " says meow"
;;

say (Cat "Tom") ;; (* “Tom says meow". *)
```

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)) ;;
```

TODOs by next lecture

- Get familiar with OCaml
- Come to the discussion session if you are new to OCaml