Say hello to OCaml

```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);
    }
}
```

Why readability matters...

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

Say hello to OCaml

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

Plan (next 4 weeks)

1. Fast forward
   - Rapid introduction to what’s in OCaml
2. Rewind
3. Slow motion
   - Go over the pieces individually

History, Variants

“Meta Language”
- Designed by Robin Milner @ Edinburgh
- Language to manipulate Theorems/Proofs
- Several dialects:
  - Standard” ML (of New Jersey)
    - Original syntax
  - “O’Caml: The PL for the discerning hacker”
    - French dialect with support for objects
    - State-of-the-art
    - Extensive library, tool, user support
    - (.NET)

ML’s holy trinity

- 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?

---

Base type: Integers

<table>
<thead>
<tr>
<th>2</th>
<th>2+2</th>
<th>2 * (9+10)</th>
<th>2 * (9+10) -12</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>38</td>
<td>26</td>
</tr>
</tbody>
</table>

Complex expressions using “operators”:
- \(+, -, *, div, mod\)

---

Base type: Strings

```
"ab"   ->  "ab"
"ab" ^ "xy"  ->  "abxy"
```

Complex expressions using “operators”:
- Concatenation ^

---

Base type: Booleans

```
true  ->  true
false ->  false
1 < 2  ->  true
"aa" = "pq"  ->  false
("aa" = "pq") && (1<2)  ->  false
("aa" = "pq") || (1<2)  ->  true
```

Complex expressions using “operators”:
- “Relations”: \(=, <, <=, >=\)
- \&\&\&, ||, not

---

Type Errors

```
(2+3) || ("a" = "b")

"pq" ^ 9

(2 + "a")
```

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

---

Complex types: Product (tuples)

```
(2+2 , 7>8);
```

(4,false)
Complex types: Product (tuples)

(9-3,"ab"^^"cd",(2+2 , ?>8))  (6,"abcd",(4,false))

(int * string * (int * bool))

• Triples, ...
• Nesting:
  - Everything is an expression, nest tuples in tuples

Complex types: Lists

[1; "pq"];

All elements must have same type

Complex types: Lists

List operator “Cons” ::

1::[];  [1]
1::[2];  [1;2;3]
"a"::["b";
"c"];
["a";"b";"c"]

Can only “cons” element to a list of same type

1::["b"; "cd"];

Complex types: Lists

List operator “Append” @

[1;2;3]@[3;4;5];  [1;2;3;4;5] int list
["a";0["b"]];  ["a";"b"] string list
[1]@[1];  [1] string list

Can only append two lists

1 @ [2;3];

... of the same type

[1] @ ["a";"b"];

Complex types: Lists

List operator “head” hd

hd [1;2];  1 int
hd (["a";0["b"]]);  ["a"] string

Only take the head a nonempty list

hd [];}
Complex types: Lists

List operator “tail” \( \text{tl} \)

\[
\begin{align*}
\text{tl} \ [1;2;3] &\rightarrow [2;3] \quad \text{int list} \\
\text{tl} \ [\{a\}@[b]] &\rightarrow [b] \quad \text{string list}
\end{align*}
\]

Only take the tail of nonempty list \( \text{tl} []; \)

Recap: Tuples vs. Lists ?

What’s the difference?

- Tuples:
  - Different types, but fixed number:
    - pair = 2 elts
    - triple = 3 elts
  - Syntax:
    - Tuples = comma
    - Lists = semicolon

- Lists:
  - Same type, unbounded number:
    - \([3;4;5;6;7]\) \quad \text{int list}

Recap: Tuples vs. Lists ?

So far, a fancy calculator...

... what do we need next?

Variables and bindings

\[
\text{let } x = e; \\
\text{"Bind the value of expression } e \text{ to the variable } x" \\
\]

Later declared expressions can use \( x \)

- Most recent “bound” value used for evaluation

\[
\begin{align*}
\text{# let } x = 2+2;; \\
\text{val } x : \text{int} = 4 \\
\text{# let } y = x * x * x;; \\
\text{val } y : \text{int} = 64 \\
\text{# let } z = [x;y;x+y];; \\
\text{val } z : \text{int list} = [4;64;68] \\
\text{#}
\end{align*}
\]
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

Catches many bugs due to typos

Local bindings

... for expressions using “temporary” variables

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

- tempVar is bound only inside expr body
- Not visible (“in scope”) outside

Binding by Pattern-Matching

Simultaneously bind several variables

```
# let (x,y,z) = (2+3,"a" ^ "b", 1::[2]);;
```

val x : int = 5
val y : string = "ab"
val z : int list = [1;2]

Binding by Pattern-Matching

But what of:

```
# let h::t = [1;2;3];;
```

Warning P: this pattern-matching not exhaustive.

val h : int = 1
val t : int list = [2,3]

```
Reason: this pattern-matching not exhaustive.
```

Why is it whining?

```
# let h::t = [];
```

Exception: Match Failure

```
# let l = [1;2;3];
```

val l = [1;2;3]

```
val h::t = l;
```

Warning: Binding not exhaustive

val h = 1 : int
val t = [2,3] : int

In general l may be empty (match failure!)

Another useful early warning

Next: functions, but remember...

Expression ➔ Value ➔ Type

A function is ...

Complex types: Functions!

Parameter (formal)

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

fn

```
int -> int
```

How a call (“application”) is evaluated:

1. Evaluate argument
2. Bind formal to arg value
3. Evaluate “Body expr”
A Problem

Can functions only have a single parameter?

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

How a call ("application") is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate "Body expr"

A Solution: Simultaneous Binding

Can functions only have a single parameter?

```
fun (x,y) -> x<y;
```

Another Solution

Whoa! A function can return a function

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

A shorthand for function binding

```
# let neg = fun f -> fun x -> not (f x);
# let neg f x = not (f x);
```

Put it together: a “filter” function

```
- 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 = fn
```

- let list1 = [1,3,12,4,7,2,10];
- let list2 = [3,12,7,10]
- filter even list1; val it : int list = [1,2,10]
- filter even list2; val it : int list = [12,4,2,10]
Put it together: a “partition” function

```ocaml
# 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], [1,2,10])
# partition even list1;
val it : (int list * int list) = ([12,4,2,10], [1,31,7])
```

A little trick ...

```ocaml
# 2 <= 3;;
val it : bool = true
# "ba" <= "ab";;
val it : bool = false
# let lt = (<) ;;
val it : 'a -> 'a -> bool = fn
# lt 2 3;;
val it : bool = true;
# lt "ba" "ab" ;;
val it : bool = false;
```

Put it together: a “quicksort” function

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