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);
}
}
```

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

Why readability matters...

```j
sort=:((=:@(<#[),~#[]),~#>[]=~@#)^:(1:<#)
```

Quicksort in J

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

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

Base type: Strings

- Concatenation \(^\text{[why the quotes ?]}\)

Complex expressions using “operators”:

Base type: Booleans

- “Relations”: \( =, <, <=, >= \)
- \&\&, ||, not

Complex expressions using “operators”:

Type Errors

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

Complex types: Product (tuples)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (2+2, 7&gt;8) )</td>
<td>( (4,\text{false}) )</td>
</tr>
<tr>
<td>“pq” ^ 9</td>
<td>int ^ bool</td>
</tr>
</tbody>
</table>
Complex types: Product (tuples)


Complex types: Lists

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


Complex types: Lists

- Unbounded size
- Can have lists of anything
- But...


Complex types: Lists

- All elements must have same type


Complex types: Lists

List operator “Cons” ::

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


Complex types: Lists

List operator “Append” @

Can only append two lists of the same type


Complex types: Lists

List operator “head” hd

Only take the head a nonempty list
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
- Lists:
  - Same type, unbounded number:
  - Syntax:
    - Tuples = comma
    - Lists = semicolon

Recap: Tuples vs. Lists?

What’s the difference?

- Tuples:
  - Different types, but fixed number:
    - (3, "abcd")
    - (int * string)
  - pair = 2 elts
  - triple = 3 elts
- Lists:
  - Same type, unbounded number:
  - Syntax:
    - Tuples = comma
    - Lists = semicolon

Variables and bindings

\[
\begin{align*}
\text{let} \ x & = e \\
\text{“Bind the value of expression} \ e \ \text{to the variable} \ x”
\end{align*}
\]

let \( x = e; \)

Variables and bindings

Later declared expressions can use \( x \)

- Most recent “bound” value used for evaluation

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

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

```ocaml
# 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

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

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

Binding by Pattern-Matching

Simultaneously bind several variables

```ocaml
# 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:

```ocaml
# let h::t = [1;2;3];;
Warning P: this pattern-matching not exhaustive.
val h : int = 1
val t : int list = [2,3]
```

Why is it whining?

```ocaml
# let h::t = [];
Exception: Match Failure
# let l = [1;2;3];
val l : int list = [1;2;3]
- val h::t = l;
Warning: Binding not exhaustive
val h : int
val t : int
```

In general `l` may be empty (match failure!)

Another useful early warning

Next: functions, but remember...

```
Expression Value

Parameter (formal) Body Expr

fun x -> x+1;
```

A function is ...

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

How a call (“application”) is evaluated:

1. Evaluate argument
2. Bind formal to arg value
3. Evaluate “Body expr”
Can functions only have a single parameter?

### A Problem

**Parameter (formal)**

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

**Body Expr**

```
int -> int
```

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

### A Solution: Simultaneous Binding

**Parameter (formal)**

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

**Body Expr**

```
(int * int) -> bool
```

Can functions only have a single parameter?

### Another Solution

**Parameter (formal)**

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

**Body Expr**

```
int -> (int -> bool)
```

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 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
```

### And how about...

**Parameter (formal)**

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

**Body Expr**

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

A function can also take a function argument

````
# let neg = fun f -> fun x -> not (f x);
val neg : int -> int -> bool = fn
# let is5gte = neg is5lt;
val is5gte : int -> bool = fn;
# is5gte 10;
val it : bool = false;
# is5gte 2;
val it : bool = true;
(*...odd, even...*)
```

### A shorthand for function binding

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

### Put it together: a "filter" function

A shorthand for function binding

````
# 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];;
# filter is5lte list1 ;
val it : int list = [3,12,7,10]
# filter is5gte list1 ;
val it : int list = [1,2,10]
# filter even list1 ;
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
```

```ocaml
# let list1 = [1,31,12,4,7,2,10];
```

```ocaml
# 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))
```