OCaml

1. Introduction
   • Rapid introduction to what’s in OCaml

2. Focus on Features Individually as Needed as Semester Progresses

*Notes from Sorin Lerner at UCSD*

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

Expression ➞ Value ➞ Type

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

Complex expressions using “operators”: (why the quotes?)
• +, -, *
• div, mod

Base type: Strings

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ab”</td>
<td>“ab”</td>
</tr>
<tr>
<td>“ab” ^ “xy”</td>
<td>“abxy”</td>
</tr>
</tbody>
</table>

Complex expressions using “operators”: (why the quotes?)
• 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

Complex types: Product (tuples)

(2+2 , 7>8); → (4,false)

int * bool

Complex types: Lists

[| ]; [] "a" list
[1;2;3]; [1;2;3] list list
[1+1;2+2;3+3;4+4]; [2;4;6;8] list list
["aa";"bb";"cc";"dd"]; ["a";"b";"cd"] string list
[(1,"ab");(7,"cd")] [(1,"ab");(7,"cd")] (int*string) list
[(1;2);3];[(4;5);6]] [(1;2);3];[(4;5);6]] (int list) list

Type Errors

(2+3) || ("aa" = "pq")

"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)

(9-3,"ab"^"cd",(2+2 , 7>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

• Unbounded size
• Can have lists of anything
• But...
Complex types: Lists

List operator “Cons” ::

- `1 :: [2]`; int list
- `1 :: ["a","b","c"]` string list

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

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

Complex types: Lists

List operator “Append” @

- `[1;2] @ [3;4;5]`; int list
- `["a";"b"]; @ ["a","b"]` string list

Can only append two lists... of the same type

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

Complex types: Lists

List operator “head” hd

- `hd [1;2];` int
- `hd ["a";"b";"c"];` string

Only take the head a nonempty list

```
hd [];
```

Complex types: Lists

List operator “tail” tl

- `tl [1;2;3];` int
- `tl ["a";"b";"c"];` string

Only take the tail of nonempty list

```
tl [];
```

Recap: Tuples vs. Lists?

What’s the difference?

- **Tuples:**
  - Different types, but fixed number:
    - `(int, string)`
    - pair = 2 elts
    - `(int, string, real)`
    - triple = 3 elts

- **Lists:**
  - Same type, unbounded number:
    - `[3;4;5;6;7]` int list

- **Syntax:**
  - Tuples = comma
  - Lists = semicolon
So far, a fancy calculator...

... what do we need next?

Variables and bindings

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

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

```
# 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]
```

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]
```

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]
```

- \( \text{tempVar} \) is bound only inside expr body from int
- 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:

```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 = [1;2;3]: list
- val h::t = l;
WARNING: Binding not exhaustive
val h : int
val t = [2,3] : int
```

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

Another useful early warning

---

### Complex types: Functions!

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>x+1</code></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>x+1</code></td>
</tr>
</tbody>
</table>

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

Can functions only have a single parameter?

---

### A Solution: Simultaneous Binding

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(x,y)</code></td>
<td><code>x&lt;y</code></td>
</tr>
</tbody>
</table>

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

Whoa! A function can return a function

### Another Solution

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>fun y -&gt; x&lt;y</code></td>
</tr>
</tbody>
</table>

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

Can functions only have a single parameter?
And how about…

A function can also take a function argument

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

```
val lt : 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;
```

“match” statement

If arg “matches” this pattern, use this body

```
let mystery l =
  match l with
  | [] -> 0
  | (h::t) -> h
```

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 = [1,31,12,4,7,2,10];
```

```
# filter is5lt list1 
```

```
val it : int list = [31,12,7,10]
```

```
# 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

```
# 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 …

```
# 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;
```

A shorthand for function binding

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

```
# let neg f 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;
```
Put it together: a “quicksort” function

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