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

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>Expression</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
- true
- false

Complex expressions using "operators":
- "Relations": =, <, <=, >=
- &&, ||, not

**Type Errors**

- (2+3) || ("aa" = "bb")
- "pq" ^ 9
- (2 + "aa")

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)
- int * bool

**Complex types: Lists**

- "a" list
- [1;2;3]; [1;2:3]; int list
- (2+1;2+2;3+3;4+4); [2;4;6;8]; int list
- "a"; "b"; "cd"; string list
- ("ab","cd","ef"); ([1,"ab"],[7,"cd"]); [int,string] list
- (1;2;3);[4;5;6]; int list

- All elements must have same type
- Unbounded size
- Can have lists of anything
- But...

- [1; "pq"];

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

List operator “Cons” ::

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

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

Complex types: Lists

List operator “Append” @

Can only append two lists of the same type

1 @ [2; 3];

Complex types: Lists

List operator “head” hd

Only take the head a nonempty list

hd [];

Complex types: Lists

List operator “tail” tl

Only take the tail of nonempty list

tl [];

Recap: Tuples vs. Lists ?

What’s the difference?

- Tuples:
  - Different types, but fixed number:
    - (3, "abcd") (int * string)
    - pair = 2 elts
    - (3, "abcd", (3.5, 4.2)) (int * string * (real * 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

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
```

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 h::t = l;
- Warning: Binding not exhaustive
val h : int = 1
val t : int list = [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>fun x -&gt; x+1</code>;</td>
<td><code>fn</code></td>
</tr>
</tbody>
</table>

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

A Problem

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>fun (x,y) -&gt; x&lt;y</code>;</td>
<td><code>fn</code></td>
</tr>
</tbody>
</table>

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

A Solution

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

Whoa! A function can return a function

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

Next: functions, but remember ...

Expression → Value

Type

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

A function is ...

Parameter (formal)

Body (Expr)
And how about...

A function can also take a function argument

```
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;
```

“match” statement

```
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
```
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))
;;
```