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

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

Base type: Strings

```
"ab"
"ab" ^ "xy" ➔ "abxy"
```

Complex expressions using “operators”:
- Concatenation ^
Base type: Booleans

<table>
<thead>
<tr>
<th>Base type: Booleans</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>1 &lt; 2</td>
</tr>
<tr>
<td>&quot;aa&quot; = &quot;pq&quot;</td>
</tr>
<tr>
<td>(&quot;aa&quot; = &quot;pq&quot;) &amp;&amp; (1&lt;2)</td>
</tr>
<tr>
<td>(&quot;aa&quot; = &quot;pq&quot;) &amp;&amp; (1&lt;2)</td>
</tr>
</tbody>
</table>

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

Type Errors

<table>
<thead>
<tr>
<th>Untypable expression is rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2+3)</td>
</tr>
<tr>
<td>&quot;pq&quot; ^ 9</td>
</tr>
<tr>
<td>(2 + &quot;a&quot;)</td>
</tr>
</tbody>
</table>

Complex types: Product (tuples)

<table>
<thead>
<tr>
<th>Complex types: Product (tuples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2+2 , ?&gt;8);</td>
</tr>
<tr>
<td>(4,false)</td>
</tr>
</tbody>
</table>

Complex types: Lists

<table>
<thead>
<tr>
<th>Complex types: Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[1;2;3];</td>
</tr>
<tr>
<td>[1;2;3],3;4+4);</td>
</tr>
<tr>
<td>[&quot;aa&quot;=&quot;b&quot;; &quot;c&quot;=&quot;d&quot;];</td>
</tr>
<tr>
<td>[(1,&quot;ab&quot;):(7,&quot;c&quot;)];</td>
</tr>
<tr>
<td>(int*string)*list</td>
</tr>
</tbody>
</table>

But...

All elements must have same type
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 1@[2;3];
... of the same type [1]@[“a”;“b”];

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

```plaintext
let x = e;

“Bind the value of expression e
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]
```

Variables and bindings

Later declared expressions can use `x`

- Most recent “bound” value used for evaluation

```plaintext
# 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!

```plaintext
# 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

```plaintext
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

```plaintext
# 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:

```plaintext
# 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?

```plaintext
# let h:t = [ ];
Exception: Match_failure
# let l = [1;2;3]; list
- val h:t = [ ];
Warning: Binding not exhaustive
val h = 1 : int
val t = [2,3] : list
```

In general I may be empty (match failure!)

Another useful early warning

Complex types: Functions!

Parameter (formal)  Body Expr
```
fun x -> x+1;
```

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

Parameter (formal)  Body Expr
```
fun (x,y) -> x<y;
```

Can functions only have a single parameter?

Another Solution

Parameter (formal)  Body Expr
```
fun x -> fun y -> x<y;
```

Whoa! A function can return a function

```plaintext
# 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…

A function can also take a function argument

Put it together: a “filter” function

Put it together: a “partition” function

A little trick…

Put it together: a “quicksort” function