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

Expression: 2
2+2
2 * (9+10)
2 * (9+10) +12

Value: 2
4
38
-12
26

Type: int

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

Base type: Strings

Expression: “ab”
“ab” ^ “xy”

Value: “abxy”

Type: string

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

<table>
<thead>
<tr>
<th>True</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>1 &lt; 2</td>
<td>true</td>
</tr>
<tr>
<td>&quot;aa&quot; = &quot;pq&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

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

**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)**

- (2+2 , 7>8)
- (4,false)
- int * bool

**Complex types: Lists**

- [ ]; []
- [1;2;3]; [1:2:3]
- (1+1,2+2,3+3,4+4)]; [2:4:6:8]
- ["aa";"bb"; "cd";"cd"]; ["ab","bc","cd"]
- [(1,"aa";"bb"); (3+4,"cc")]; [(1,"ab"),(7,"cc")]
- [[1];[2];[3]]; [[1];[2];[3];[4;5;6]]

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

All elements must have same type
Complex types: Lists

List operator “Cons” ::

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

Recap: Tuples vs. Lists?

What’s the difference?

• Tuples:
  - Different types, but fixed number:
    - (3, "abcd")
    - (3, "abcd", (3,5,4.2))
  - pair = 2 elts
  - triple = 3 elts

• Lists:
  - Same type, unbounded number:
    - [3;4;5;6;7]

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

... what do we need next?

Variables and bindings

\[
\text{let } x = \text{e};
\]

“Bind the value of expression e to the variable x”

\[
\# \text{ let } x = 2+2;;
\text{val } x : \text{int} = 4
\]

\[
\# \text{ let } x = 2+2;;
\text{val } x : \text{int} = 4
\]

\[
\# \text{ let } y = x * x * x;;
\text{val } y : \text{int} = 64
\]

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

Variables and bindings

Later declared expressions can use x

- Most recent “bound” value used for evaluation

\[
\# \text{ let } x = 2+2;;
\text{val } x : \text{int} = 4
\]

\[
\# \text{ let } y = x * x * x;;
\text{val } y : \text{int} = 64
\]

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

Variables and bindings

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

\[
\# \text{ let } p = a + 1;;
\]

\[
\text{Characters 8-9:}
\]

\[
\text{let p = a + 1 ;;}
\]

\[
\text{Unbound value } a
\]

Catches many bugs due to typos

Variables and bindings

Local bindings

... for expressions using “temporary” variables

\[
\text{let tempVar = x + 2 * y in tempVar * tempVar};;
\]

- tempVar is bound only inside expr body from in

\[
\text{17424 int}
\]

- Not visible (“in scope”) outside

Binding by Pattern-Matching

Simultaneously bind several variables

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

\[
\text{val x : int} = 5
\]

\[
\text{val y : string} = "ab"
\]

\[
\text{val z : int list} = [1;2]
\]
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 hl: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.
```

Complex types: Functions!

```plaintext
fun x -> x+1;;
fun x,y -> x<y;
fun x -> fn y -> x<y;
```

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

A Problem

```plaintext
fun x -> x+1;;
fun x -> fn y -> x<y;
```

Can functions only have a single parameter?

A Solution: Simultaneous Binding

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

Next: functions, but remember ...

Expression ➔ Value ➔ Type

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

A function is ...

Parameter (formal) ➔ Expr ➔ Body

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

Another Solution

```plaintext
fun x -> fn y -> x<y;
fun (x,y) -> (int * int) -> bool
```

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