OCaml

The PL for the discerning hacker.
ML Flow

1. Enter expression
2. ML infers a type $\tau$
3. ML crunches expression down to a value
4. Value guaranteed to have type $\tau$

Typing $\Rightarrow$ Eval Always Works
## Complex types: Lists

- **Unbounded size**
- Can have lists of anything (e.g. lists of lists)
- But...

<table>
<thead>
<tr>
<th>[]</th>
<th>[]</th>
<th>'a list</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1;2;3]</td>
<td>[1;2;3]</td>
<td>int list</td>
</tr>
<tr>
<td>[1+1;2+2;3+3;4+4]</td>
<td>[2;4;6;8]</td>
<td>int list</td>
</tr>
<tr>
<td>[“a”;“b”; “c”^^“d”]</td>
<td>[“a”;“b”; “cd”]</td>
<td>string list</td>
</tr>
<tr>
<td>[ (1,“a”^^“b”); (3+4,“c”)]</td>
<td>[(1,“ab”);(7,“c”)]</td>
<td>(int*string) list</td>
</tr>
<tr>
<td>[[1];[2;3];[4;5;6]]</td>
<td>[[1];[2;3];[4;5;6]]</td>
<td>(int list) list</td>
</tr>
</tbody>
</table>
Complex types: Lists

All elements **must have same type**

```
[1; "pq"];
```
Question 1

Which of these causes a type error?

(a) [1; 2; 3]
(b) [“1”, 2, 3]
(c) “[1; 2; 3]”
(d) (1, 2, 3)
(e) [“1”; 2; 3]
Complex types: Lists

List operator “Cons” ::

```
1::[];
1::[2;3];
"a"::["b";"c"];
```

```
[1]
[1;2;3]
[“a”;“b”;“c”]
```

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

```
1::[“b”;“cd”];
```
Lists: Construct

Nil operator

\[
[\ ] \quad \Rightarrow \quad [\ ] \quad \Rightarrow \quad [\ ]
\]

Cons operator

\[
1: \cdot [2;3] \quad \Rightarrow \quad 1: \cdot [2;3]
\]

\[
e_{1}: T \quad e_{2}: T \ \text{list} \quad \Rightarrow \quad e_{1}: e_{2} : T \ \text{list}
\]

\[
e_{1}=>v_{1} \quad e_{2}=>v_{2} \quad e_{1}: e_{2} \Rightarrow v_{1}: v_{2}
\]
Complex types: Lists

List operator “Append”  

[1;2]@[3;4;5];  
[1;2;3;4;5]  
int list

[“a”]@[“b”];  
[“a”;“b”]  
string list

[]@[1];  
[1]  
int list

Can only append two lists  

1@[2;3];  
[1]@[“a”;“b”];

... of the same type
Complex types: Lists

List operator “head” \( \text{hd} \)

\[
\text{hd [1;2];}
\]

\[
\text{hd ["a";"b"];}
\]

Only take the head a nonempty list

\[
\text{hd []};
\]
Complex types: Lists

List operator “tail” \( \text{tl} \)

\[
\begin{align*}
\text{tl} & \quad [1;2;3]; \quad [2;3] \\
\text{tl} & \quad \text{[“a”]@[“b”]}; \quad \text{[“b”]} \\
\text{tl} & \quad []; \quad \text{int list} \\
\text{tl} & \quad \text{string list}
\end{align*}
\]

Only take the tail of nonempty list \( \text{tl} \quad []; \)
Question 2: What is result of?

\[(\text{hd} \; [[];[1;2;3]]) = (\text{hd} \; [[];[“a”]])\]

(a) Syntax Error
(b) true : bool
(c) false : bool
(d) Type Error (hd)
(e) Type Error (=)
Lists: Deconstruct

**Head**

\[
\begin{align*}
    e & : T \text{ list} \quad e \Rightarrow v_1::v_2 \\
    \text{hd } e & : T \\
    \text{hd } e & \Rightarrow v_1
\end{align*}
\]

**Tail**

\[
\begin{align*}
    e & : T \text{ list} \quad e \Rightarrow v_1::v_2 \\
    t\text{l } e & : T \text{ list} \\
    t\text{l } e & \Rightarrow v_2
\end{align*}
\]

\[
(\text{hd } [[[]];[1;2;3]]) = (\text{hd } [[[]];[“a”]])
\]

\[
\begin{align*}
    e_1 &: T \\
    e_2 &: T \\
    e_1 & = e_2 : \text{ bool}
\end{align*}
\]

**int list**

**string list**
Recap: Tuples vs. Lists?

What’s the difference?

• Tuples:
  - Different types, but fixed number:
    - pair = 2 elts
      - (3, “abcd”) (int * string)
    - triple = 3 elts
      - (3, “abcd”, (3.5, 4.2)) (int * string * (float * float))

• 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?
So far, a fancy calculator...

Branches
Question 3: What is result of?

\[
\text{if } (1 < 2) \text{ then } \text{true} \text{ else } \text{false}
\]

(a) Syntax Error
(b) true
(c) false
(d) Type Error
Question 4: What is result of?

if (1 < 2) then [1;2] else 5

(a) Syntax Error
(b) [1;2]
(c) 5
(d) Type Error
If-then-else expressions

- Then-subexp, Else-subexp must have same type!
  - Equals type of resulting expression

\[
\begin{align*}
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\end{align*}
\]

\[
\begin{align*}
\text{if } 1>2 \text{ then } [1,2] \text{ else } [] = (\text{if } 1<2 \text{ then } [] \text{ else } ["a"])
\end{align*}
\]
If-then-else expressions

\[
\text{if } (1 < 2) \text{ then } [1;2] \text{ else } 5
\]

\[
\text{if } \text{false} \text{ then } [1;2] \text{ else } 5
\]

- then-subexp, else-subexp must have same type!
  - ...which is the type of resulting expression

\[
e1 : \text{bool} \quad e2 : T \quad e3 : T
\]

\[
\text{if } e1 \text{ then } e2 \text{ else } e3 : T
\]
So far, a fancy calculator...

Variables
Question 5: I got this `@` prompt

```plaintext
# [x+x; x*x] ;;
- : int list = [20; 100]
```

What had I typed before?

(a) `x = 10;`
(b) `int x = 10;`
(c) `x == 10;`
(d) `let x = 10;`
(e) `x := 10;`
Variables and bindings

```
let x = e;;
```

“Bind the value of expression \( e \) to the variable \( x \)”

```ml
# let x = 2+2;;
val x : int = 4
```
Variables and bindings

Later declared expressions can use \( x \)
- Most recent “bound” value used for evaluation

\[
# \textbf{let } x = 2+2;; \\
\text{val } x : \text{int} = 4 \\
# \textbf{let } y = x \times x \times x;; \\
\text{val } y : \text{int} = 64 \\
# \textbf{let } z = [x;y;x+y];; \\
\text{val } z : \text{int list} = [4;64;68] \\
#
\]
Variables and bindings

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

```haskell
# 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

• tempVar is bound only inside expr body
  from in ... 
• Not visible (“not in scope”) outside

17424 int
Question 6: What is result of?

\[
\text{let } x = 10 \text{ in } \\
(\text{let } z = 10 \text{ in } x + z) + z
\]

(a) Syntax Error
(b) 30
(c) Unbound Error -- x
(d) Unbound Error -- z
(e) Type Error
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 XS = [1;2;3];
val xs = [1;2;3]: list
- val h::t = xs;
Warning: Binding not exhaustive
val h = 1 : int
val t = [2;3] : int
```

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

Another useful early warning
Binding by Pattern-Matching

But what of:

```ml
let h::t = [1;2;3];;
```

Warning P: this pattern-matching not exhaustive.

```ml
val h : int = 1
val t : int list = [2;3]
```

Why is it whining?

```ml
let h::t = [];;
```

Exception: Match_failure

```ml
let XS = [1;2;3];;
val xs = [1;2;3]: list
```

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

Another useful early warning

**NEVER USE PATTERN MATCHING LIKE THIS**

```ml
let h::t = ...
```

**ALWAYS USE THIS FORM INSTEAD**

```ml
match l with ...
```

(coming up soon, but this is important)
Functions
Functions up now, remember ...

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

A function is a value!
Complex types: Functions!

Parameter (formal) | Body Expr
--- | ---

```ml
fun x -> x+1
```

```ml
# let inc = fun x -> x+1 ;
val inc : int -> int = fn
# inc 0;
val it : int = 1
# inc 10;
val it : int = 11
```
Functions only have ONE parameter ?!

Parameter (formal)

Body Expr

fun x -> x+1

fn

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)       Body Expr

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

(int * int) -> bool

Functions only have ONE parameter?

How a call (“application”) is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate “Body expr”
Another Solution (“Currying”) 

Whoa! A function can return a function

```ocaml
# let lt = fun x -> fun 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;
```
Question 7: What is result of?

\((\text{fun } \ x \ \rightarrow \ \text{not } \ x)\)

(a) Syntax Error
(b) \(<\text{fun}> : \text{int} \ \rightarrow \ \text{int}\)
(c) \(<\text{fun}> : \text{int} \ \rightarrow \ \text{bool}\)
(d) \(<\text{fun}> : \text{bool} \ \rightarrow \ \text{int}\)
(e) \(<\text{fun}> : \text{bool} \ \rightarrow \ \text{bool}\)
And how about...

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body</th>
<th>Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun f -&gt;</td>
<td>fun x -&gt; not(f x);</td>
<td>fn</td>
</tr>
</tbody>
</table>

A function can also take a function argument

```ocaml
# 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;
(*...odd, even ...*)
```
Question 8: What is result of?

\[(\text{fun } f \rightarrow (\text{fun } x \rightarrow (f \, x) + x))\]

(a) Syntax Error
(b) Type Error
(c) \(<\text{fun}> : \text{int} \rightarrow \text{int} \rightarrow \text{int}\>
(d) \(<\text{fun}> : \text{int} \rightarrow \text{int}\>
(e) \(<\text{fun}> : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}\>
A shorthand for function binding

```ocaml
# 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 “filter” function

If arg “matches” this pattern... ...then use this Body Expr

```ml
- let rec filter f xs =  
  match xs with  
  | []     -> []  
  | (x::xs')-> if f x 
    then x::(filter f xs') 
    else (filter f xs');; 

val filter : ('a->bool)->'a list->'a lisi) = fn
```

```ml
# 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;4;2]  
# 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;

# let is5Lt = lt 5;
val is5lt : int -> bool = fn;
# is5lt 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
Put it together: a “quicksort” function

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

Now, let's begin at the beginning …