Any questions

 Say hello to OCaml

 Why readability matters...

 Plan (next 4 weeks)

 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

- 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>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ab”</td>
<td>string</td>
</tr>
<tr>
<td>“ab” ^ “xy”</td>
<td></td>
</tr>
</tbody>
</table>

Complex expressions using “operators”: * (why the quotes?)
- Concatenation ^

Base type: Booleans

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>1 &lt; 2</td>
<td>true</td>
</tr>
<tr>
<td>“aa” = “pq”</td>
<td>false</td>
</tr>
<tr>
<td>(“aa” = “pq”) &amp;&amp; (1&lt;2)</td>
<td>false</td>
</tr>
<tr>
<td>(“aa” = “pq”) &amp;&amp; (1&lt;2)</td>
<td>true</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);  \rightarrow  (4, false)

(int * bool)

Complex types: Product (tuples)

(9-3, "ab"^"cd", (2+2, 7>8));  \rightarrow  (6, "abcd", (4, false))

(int * string * (int * bool))

- Triples,
- Nesting:
  - Everything is an expression, nest tuples in tuples

Complex types: Lists

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

Complex types: Lists

All elements must have same type

List operator “Cons” ::

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

1 :: ["b"; "cd”];

List operator “Append” @

Can only append two lists of the same type

1 @ [2;3];

[1] @ [“a”;“b”];
Complex types: Lists

List operator “head” \( \text{hd} \)

- \( \text{hd} [1;2]; \)
- \( \text{hd} (["a"]@["b"]); \)

Only take the head of a nonempty list \( \text{hd} []; \)

Complex types: Lists

List operator “tail” \( \text{tl} \)

- \( \text{tl} [1;2;3]; \)
- \( \text{tl} (["a"]@["b"]); \)

Only take the tail of a nonempty list \( \text{tl} []; \)

Recap: Tuples vs. Lists?

What’s the difference?

- Tuples:
  - Different types, but fixed number:
    - Pair = 2 elts
    - Triple = 3 elts
  - Syntax:
    - Tuples = comma

- Lists:
  - Same type, unbounded number:
  - Syntax:
    - 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 \( x = 2+2; \) \\
\( \text{val } x : \text{ int} = 4 \)
Variables and bindings

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

```ocaml
# 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!

```ocaml
# 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

```ocaml
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

```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 = [1];
Exception: Match failure
# let l = [1;2;3];
val l : int list = [1;2,3];
- val h::t = 1;
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

---

Next: functions, but remember ...

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

A function is ...
Complex types: Functions!

Parameter
(formal)
Body
Expr
\[ \text{fun } x \rightarrow x+1 \]
\[ \text{fn} \]

# 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

Parameter
(formal)
Body
Expr
\[ \text{fun } x \rightarrow x+1 \]
\[ \text{fn} \]

Can functions only have a single parameter?

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
\[ \text{fun } (x,y) \rightarrow x<y \]
\[ \text{fn} \]

A function can return a function

# let lt = fun x -> fn y -> x < y ;
val lt : int -> int -> bool = fn
# let is5lt = lt 5;
val it : bool = true;
# is5lt 10;
val it : bool = false;

Whoa! A function can return a function

# let neg = fun f -> fun x -> not (f x);
val neg : int -> int -> bool = fn
# let neg is5lt = neg is5lt;
val it : bool = true;
(*…odd, even …*)

A shorthand for function binding

# let neg = fun f -> fun x -> not (f x);
val neg : int -> int -> bool = fn
# let neg is5lt = neg is5lt;
val is5gte : int -> bool = fn
# is5gte 10;
val it : bool = false;
# is5gte 2;
val it : bool = true;

And how about…

Parameter
(formal)
Body
Expr
\[ \text{fun } f \rightarrow \text{fun } x \rightarrow \text{not}(f x) \]
\[ \text{fn} \]

Can functions only have a single parameter?

A function can also take a function argument

# let neg = fun f -> fun x -> not (f x);
val it : int -> int -> bool = fn
# let is5gte = neg is5lte;
val it : bool = true;
("a -> bool") -> ("a -> bool")

# let neg = fun f -> fun x -> not (f x);
val it : int -> int -> bool = fn
# let neg x = not (x);
val neg : int -> bool = fn

A function can also take a function argument
Put it together: a “filter” function

If arg “matches”...then use

this pattern...

```ocaml
- 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
```

If arg “matches”…then use this Body Expr

# 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

```ocaml
- let partition f l = (filter f l, filter (neg f) l);

val partition : ('a -> bool) -> 'a list -> 'a list * 'a list = fn
```

```ocaml
# 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])
```

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));;

val sort : 'a list -> 'a list = fn
```

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

val sort : 'a list -> 'a list = fn
```

A little trick …

```ocaml
# 2 <= 3;; ...
val it : bool = true
# "ba" <= "ab";;
val it : bool = false
# let lt = (<) ;;
val lt : 'a -> 'a -> bool = fn
# lt 2 3;;
val it : bool = true;
# lt "ba" "ab" ;;
val it : bool = false;
```

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

val sort : 'a list -> 'a list = fn
```

```ocaml
# 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])
```