News

On webpage:
- Suggested HW #1
- PA #1 (due next Fri 4/8)

Please post questions to WebCT

Today: A crash course in ML contd...

Type Errors

Untypable expression is rejected
- No casting, No coercing
- Fancy algorithm to catch errors
- ML’s single most powerful feature (why?)

Complex types: Product (tuples)

(2+3) || (“a” = “b”)

(2 + “a”)
Complex types: Product (tuples)

- Triples,…  
- Nesting: 
  - Everything is an expression  
  - Nest tuples in tuples

Recap: 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 ?
  - int, bool, products (tuples), lists ...

Recap: ML’s holy trinity

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

Complex types: Lists

- Unbounded size
- Can have lists of anything (e.g. lists of lists)
- But…
Complex types: Lists

All elements must have same type

List operator “Cons” ::

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

Complex types: Lists

List operator “Append” @

Can only append two lists ... of the same type
Complex types: Lists

List operator “tail” $\text{tl}$

$$\text{tl} \ [1;2;3]; \ [2;3]$$
$$\text{tl} \ (\text{“a”}@\text{“b”}); \ [\text{“b”}]$$

Only take the tail of 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
    - (3, “abcd”) (int * string)
    - (3, “abcd”, (3.5,4.2)) (int * string * (float * float))

- **Lists:**
  - **Same** type, unbounded number:
    - Syntax:
    - Lists = semicolon
    - [3;4;5;6;7] int list

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

```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

- `tempVar` is bound only inside expr body
- Not visible (“not in scope”) outside

```ocaml
let tempVar = x + 2 * y
in tempVar * tempVar
;;
```

**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 l = [1;2;3];
val l = [1;2;3]: list
- val h::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!

Parameter (formal) | Body Expr
---|---
```ocaml
fun \(x\) \(\rightarrow\) \(x+1\);
```

```ocaml
# let inc = fun x -> x+1 ;
val inc : int -> int = fn
# inc 0;
val it : int = 1
# inc 10;
val it : int = 11
```
### A Problem

Can functions only have a single parameter?

<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></td>
</tr>
</tbody>
</table>

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

### A Solution: Simultaneous Binding

Can functions only have a single parameter?

<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></td>
</tr>
</tbody>
</table>

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

### Another Solution

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

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

A function can also take a function argument

```ocaml
# let neg = fun f -> fun x -> not(f x); val it : int -> int -> bool = fn
# let is5gte = neg is5lt;
# is5gte : int -> bool = fn
# is5gte 10;
val it : bool = true;
# is5gte 2;
val it : bool = false;
# is5gte 2;
val it : bool = true;
(*...odd, even ...*)
```
A shorthand for function binding

```ml
# 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

```ml
- let rec filter f xs =  
  match xs with  
  | []      -> [] ...  
# 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

```ml
# let partition f l = (filter f l, filter (neg f) l);
val partition : ("a list -> bool) -> "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 ...

```ml
# 2 <= 3;; ...  
val it : bool = true 
# "ba" <= "ab";;  
val it : bool = false 

# let lt = (<) ... = 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

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

Now, lets begin at the beginning ...