News

On webpage:
• Suggested HW #1
• PA #1 (due next Fri 9/8)

Please post questions to WebCT/TED

Today: A crash course in ML contd...

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...

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[]</td>
<td>'a list</td>
</tr>
<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>[&quot;a&quot;;&quot;b&quot;; &quot;c&quot;;&quot;d&quot;]</td>
<td>[&quot;a&quot;;&quot;b&quot;; &quot;cd&quot;]</td>
<td>string list</td>
</tr>
<tr>
<td>[(1,&quot;a&quot;;&quot;b&quot;); (3+4,&quot;c&quot;)];</td>
<td>[(1,&quot;ab&quot;);(7,&quot;c&quot;)];</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

List operator “Cons” ::

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

List operator “Append” @

Can only append two lists of the same type

List operator “head” hd

Only take the head a nonempty list
Complex types: Lists

List operator “tail” \( \text{tl} \)

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

int list
string list

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
    - (3, “abcd”) (int * string)
    - (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?

Variables and bindings

\[
\text{let} \ x = e; \\
\text{“Bind the value of expression } e \text{ to the variable } x”
\]

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

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

```ocaml
# let p = a + 1;;
  ^
Unbound value a
```

Catches many bugs due to typos

### Variables and bindings

**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 (“not 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:

```ml
# 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?

```ml
# 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

Next class: functions, but remember ...

Expression ➔ Value ➔ Type

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

A function is a value!

Complex types: Functions!

Parameter (formal) ➔ Body Expr ➔ fn

```
fun x -> x+1;;
```

Parameter (formal) ➔ Body Expr ➔ fn

```
fun x -> x+1;
```

```
int -> int
```

Can functions only have a single parameter?

A Problem

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?

Whoa! A function can return a function

And how about...

A shorthand for function binding

Parameter (formal) | Body Expr
---|---
fun \((x, y)\) -> \(x < y\); | fn

(int * int) -> bool

Parameter (formal) | Body Expr
---|---
fun \(x\) -> fun \(y\) -> \(x < y\); | fn

int -> (int -> bool)

Parameter (formal) | Body Expr
---|---
fun \(f\) -> fun \(x\) -> not \((f \ x)\); | fn

('a -> bool) -> ('a -> bool)

A function can also take a function argument

# 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 = true;
# is5gte 2;
val it : bool = false;
(*...odd, even ...*)
Put it together: a “filter” function

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

```ocaml
- 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 list = fn
```

# 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

```ocaml
# 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 ...

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

# let lt = (<) ;;
val it ... = lt 5;
val is5lt : int -> bool = fn:

# 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, lets begin at the beginning ...

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