A shorthand for function binding

```ocaml
# let neg = fun f -> fun x -> not (f x); ...
# let neg f x = not (f x);
val neg : ('a -> bool) -> 'a -> 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

```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

# 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

```ml
# 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 ...

```ml
# 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

```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 ...

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

• We’ve seen some base types and values:
  - Integers, Floats, Bool, String etc.

• Some ways to build up types:
  - Products (tuples), records, “lists”
  - Functions

What About More Complex Data?

• We’ve seen some base types and values:
  - Integers, Floats, Bool, String etc.

• Some ways to build up types:
  - Products (tuples), records, “lists”
  - Functions

• Design Principle: Orthogonality
  - Don’t clutter core language with stuff
  - Few, powerful orthogonal building techniques
  - Put “derived” types, values, functions in libraries

How To Build Datatypes

Three key ways to build complex types/values

1. “Each-of” types
   Value of T contains value of T1 and a value of T2

2. “One-of” types
   Value of T contains value of T1 or a value of T2

3. “Recursive”
   Value of T contains (sub)-value of same type T
Suppose I wanted ...

... a program that processed lists of attributes

- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (float)
- Alive (boolean)
- Phone (int-int)
- Email (string)

Many kinds of attributes:
- too many to put in a record
- can have multiple names, addresses, phones, emails etc.

Constructing Datatypes

t is a new datatype.
A value of type t is either:

- a value of type t1 placed in a box labeled C1
- a value of type t2 placed in a box labeled C2
- ... 
- a value of type tn placed in a box labeled Cn

Creating Values

How to create values of type attrib?

```haskell
type attrib =
  Name of string
| Age of int
| DOB of int*int*int
| Address of string
| Height of float
| Alive of bool
| Phone of int*int
| Email of string;;
```

```haskell
# let a1 = Name "Ranjit";;
val x : attrib = Name "Ranjit"
# let a2 = Height 5.83;;
val a2 : attrib = Height 5.83
# let year = 1977 ;;
val year : int = 1977
# let a3 = DOB (9,8,year) ;;
val a3 : attrib = DOB (9,8,1977)
# let a_l = [a1;a2;a3];;
val a3 : attrib list = ...
```
One-of types

- We’ve defined a “one-of” type named `attrib`
- Elements are one of:
  - string,
  - int,
  - int*int*int,
  - float,
  - bool ...

- Can create uniform `attrib` lists
- Suppose I want a function to print `attribs` ...

How to tell what’s in the box?

```markdown
match e with
  Name s -> e1
| Age i -> e2
| DOB (m,d,y) -> e3
| Address addr -> e4
| Height h -> e5
| Alive b -> e6
| Phone (a,n) -> e7
| Email e -> e8
```

Pattern-match expression: check if `e` is of the form ...

- On match:
  - value in box bound to pattern variable
  - matching result expression is evaluated

Simultaneously test and extract contents of box

match-with is an Expression

```markdown
match e with
  Name s -> e1
| Age i -> e2
| DOB (m,d,y) -> e3
| Address of string
| Height of float
| Alive of bool
| Phone of int*int
| Email of string;;
```

Type rules?

- `e1, e2, ..., en` must have same type
- Which is type of whole expression
Benefits of **match-with**

1. Simultaneous test-extract-bind
2. Compile-time checks for:
   - missed cases: ML warns if you miss a \( t \) value
   - redundant cases: ML warns if a case never matches

```ml
match e with
   C1 x1 -> e1
| C2 x2 -> e2
| ...
| Cn xn -> en
```

```ml
type t =
   C1 of t1
| C2 of t2
| ...
| Cn of tn
```

What about “Recursive” types?

Think about this! What are values of `int_list`?

```
Cons(1,Cons(2,Cons(3,Nil)))  Cons(2,Cons(3,Nil))  Cons(3,Nil)  Nil
```

Lists aren’t built-in!

Lists are derived using an elegant core!

1. Each-of
2. One-of
3. Recursive

```ml
let rec len l =
  match l with
    Nil -> 0
  | Cons(_,t) -> 1 + (len t)
```

Some functions on Lists: Length

```
let rec len l =
  match l with
    Nil -> 0
  | Cons(_,t) -> 1 + (len t)
```

Matches everything, no binding
- Must match with `Nil`
Some functions on Lists: Append

```ocaml
let rec append (l1, l2) =
  Base Expression
  Inductive Expression
  Base pattern
  Ind pattern
```

- Find the right induction strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression

Well designed datatype gives strategy

null, hd, tl are all functions ...

Bad ML style: More than aesthetics!

Pattern-matching better than test-extract:
- ML checks all cases covered
- ML checks no redundant cases
- ...at compile-time:
  - fewer errors (crashes) during execution
  - get the bugs out ASAP!

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:
- \(4.0 + 2.9 = 6.9\)
- \(3.78 - 5.92 = -2.14\)
- \((4.0 + 2.9) \times (3.78 - 5.92) = -14.766\)

Q: What's a ML datatype for such expressions?

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:
- \(4.0 + 2.9 = 6.9\)
- \(3.78 - 5.92 = -2.14\)
- \((4.0 + 2.9) \times (3.78 - 5.92) = -14.766\)

What's a ML function for evaluating such expressions?