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

```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 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 ...

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

```
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 ...

News

• PA 1 due Fri 4/8 5pm
• PA 2 out today or tomorrow

Begin at the beginning ...

Expressions (Syntax) \[\text{ Compile-time } \text{ “Static” } \]

| \text{ Compile-time } \text{ “Dynamic” } | \text{ Values (Semantics) } |
|------------------------------------------------|

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   • Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   • Of the same “type” found in step 2
### Base Types

**Base Type: int**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3</td>
<td>5</td>
</tr>
<tr>
<td>7 - 4</td>
<td>3</td>
</tr>
<tr>
<td>(2 + 3) * (7 - 4)</td>
<td>15</td>
</tr>
</tbody>
</table>

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions

**Base Type: bool**

- Comparison is built-in for all expr, values, types
  - but compared expressions must have same type
- ... except for ?
  - function values ... why?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>not (2 &lt; 3)</td>
<td>false</td>
</tr>
<tr>
<td>e1 &lt; e2</td>
<td>false</td>
</tr>
<tr>
<td>[1; 2] &lt; [3]</td>
<td>false</td>
</tr>
</tbody>
</table>
**Type Errors**

- Expressions built from sub-expressions
- Types computed from types of sub-expression
- If a sub-expression is not well-typed then whole expression is not well-typed

```plaintext
0 * (2 + "a");
```

**Complex types: Tuples**

- Can be of any fixed size
- Elements can have different types
- Tuples can be nested in other tuples

```plaintext
(9-3,"ab"^^cd",7>8) (6, "abcd", false)
```

**Complex types: Records**

- Records are tuples with named elements...

```plaintext
{ name="ranjit" ; age=33 ; pass=false }
```

**Lists: Construct**

- Nil operator
- Cons operator

```plaintext
[] => []
```

```plaintext
1::[2;3] [1;2;3]
```
Lists: Deconstruct

Head

\[ e : T \text{ list} \]
\[ \text{hd } e : T \]
\[ e \Rightarrow v_1::v_2 \]
\[ \text{hd } e \Rightarrow v_1 \]

Tail

\[ e : T \text{ list} \]
\[ t_1 e : T \text{ list} \]
\[ e \Rightarrow v_1::v_2 \]
\[ t_1 e \Rightarrow v_2 \]

Recap

Expressions (Syntax) \[\xrightarrow{\text{Evaluate}}\] Values (Semantics)

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   - Of the same “type” found in step 2

Types

- Static
- Dynamic

If-then-else expressions

\[ \text{if } (1 < 2) \text{ then } 5 \text{ else } 10 \]

\[ \text{if } (1 < 2) \text{ then } \left[ \text{"ab","cd"} \right] \text{ else } \left[ \text{"x"} \right] \]

If-then-else is also an expression!
Can use any expression in then, else branch

Recap

- Integers: +, -, *
- Floats: +, -, *
- Booleans: =, <, &&, ||, not
- Strings: ^
- Tuples, Records: #i
  - Fixed number of values, of different types
- Lists: ::, @, hd, tl, null
  - Unbounded number of values, of same type
If-then-else expressions

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

\[
e_1 : \text{bool} \quad e_2 : T \quad e_3 : T
\]

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\]

If-then-else expressions

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

\[
\begin{align*}
\text{if } 1>2 \text{ then } [1,2] \text{ else } [] & \quad \text{[]} \\
\text{if } 1<2 \text{ then } [] \text{ else } ["a"] & \quad ["a"]
\end{align*}
\]

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

Q: How to use variables in ML?
Q: How to “assign” to a variable?

```ml
# let x = 2+2;;
val x : int = 4
let x = e;;
“Bind the value of expression \(e\) to the variable \(x\)”
```

Next: Variables

Variables and Bindings
Variables and Bindings

```ml
# 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]
```

Later declared expressions can use `x` - Most recent "bound" value used for evaluation

Sounds like C/Java? NO!

Environments ("Phone Book")

How ML deals with variables
- Variables = "names"
- Values = "phone number"

Environments

"Phone book"
- Variables = "names"
- Values = "phone number"

Environments and Evaluation

ML begins in a "top-level" environment
- Some names bound

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

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. Evaluate `expr e` in current env to get value `v : t`
2. Extend env to bind `x` to `v : t`
(Repeat with next binding)
Example

```ml
# 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]

# let x = x + x ;;
val x : int = 8
```

Environments

1. Evaluate: Use most recent bound value of var
2. Extend: Add new binding at end

How is this different from C/Java’s “store”?

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

# let f = fun y -> x + y;;
val f : int -> int = fn

# let x = x + x ;;
val x : int = 8
# f 0;
val it : int = 4
```

New binding:

- No change or mutation
- Old binding frozen in `f`
**Cannot change the world**

**Cannot “assign” to variables**
- Can extend the env by adding a fresh binding
- Does not affect previous uses of variable

Environment at fun declaration **frozen** inside fun “value”
- Frozen env used to evaluate application (\(\epsilon\) ...)

**Q**: Why is this a good thing ?

**A**: Function behavior frozen at declaration
- Nothing entered afterwards affects function
- Same inputs always produce same outputs
  - Localizes debugging
  - Localizes reasoning about the program
  - No “sharing” means no evil aliasing

**Examples of no sharing**

Remember: No addresses, no sharing.
- Each variable is bound to a “fresh instance” of a value

Tuples, Lists ...

- Efficient implementation without sharing?
  - There is sharing and pointers but hidden from you

- **Compiler’s job** is to optimize code
  - Efficiently implement these “no-sharing” semantics

- **Your job** is to use the simplified semantics
  - Write correct, cleaner, readable, extendable systems

**Function bindings**

Functions are values, can bind using `val`

```ocaml
let fname = fun x -> e ;;
```

**Problem**: Can’t define recursive functions !
- `fname` is bound after computing rhs value
- no (or “old”) binding for occurrences of `fname` inside `e`

```ocaml
let rec fname x = e ;;
```

Occurrences of `fname` inside `e` bound to “this” definition

```ocaml
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```
Local bindings

So far: bindings that remain until a re-binding ("global")

Local, "temporary" variables are useful inside functions
• Avoid repeating computations
• Make functions more readable

let x = e1 in
  e2
;;

Let-in is an expression!

Evaluating let-in in env E:
1. Evaluate expr e1 in env E to get value v : t
2. Use extended E [x ↦ v : t] to evaluate e2

nested bindings

let x = 10
in
 let
  y = 20
 in
   x * y
 + x
;;

Nested bindings

Evaluating let-in in env E:
1. Evaluate expr e1 in env E to get value v : t
2. Use extended E [x ↦ v : t] to evaluate e2
Nested bindings

```ocaml
let
  x = 10
in
  let
    y = 20
  in
  x * y
;;
```

Correct Formatting

Example

```ocaml
let rec filter (f,l) = 
  if l = [] then [] 
  else 
    let h = hd l in 
    let t = filter (f, tl l) in 
    if (f h) then h::t else t
```

Nested function bindings

```ocaml
let a = 20;;

let f x = 
  let y = 10 in 
let g z = y + z in 
  a + (g x)
;;
f 0;
```

Recap

- Variables are names for values
  - Environment: dictionary/phonebook
  - Most recent binding used
  - Entries never changed, new entries added

- Environment frozen at fun definition
  - Re-binding variables cannot change a function
  - Same I/O behavior at every call
Recap

• Build complex expressions with local bindings
  - let-in expression
  - The let-binding is visible (in scope) inside in-expression
  - Elsewhere the binding is not visible

Static/Lexical Scoping

• For each occurrence of a variable, there is a unique place in program text where the variable was defined
  - Most recent binding in environment

• Static/Lexical: Determined from the program text
  - Without executing the program

• Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence

Next: Functions

Q: What’s the value of a function?