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

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

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

“Bind the value of expression e to the variable x”
Variables and Bindings

```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]
```

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 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)
Environments

“Phone book”
• Variables = “names”
• Values = “phone number”

1. Evaluate:
Find and use most recent value of variable

2. Extend:
Add new binding at end of “phone book”
Example

# 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

New binding!
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” ?**

```ocaml
# 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
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”? 

```ocaml
# 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
```
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”?

```ocaml
# 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
```

---

### Binding used to eval (f ...)###

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code,  &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

### Binding for subsequent x
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 (f …)

Q: Why is this a good thing?

```ocaml
# 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
```

Binding used to eval (f …)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Binding for subsequent x
Cannot change the world

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
Recap: Environments

“Phone book”
• Variables = “names”
• Values = “phone number”

1. Evaluate:
Find and use most recent value of variable

2. Extend: `let x = e ;;`
Add new binding at end of “phone book”
Next: Functions

Expressions \rightarrow \text{Types} \rightarrow \text{Values}
Functions

Functions are values, can bind using `let`

```
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`

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

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

```
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```
F takes a value of type T1 and returns a value of type T2
Two questions about function values:

What is the value:

1. … of a function ?

2. … of a function “application” (call) ? (e1 e2)
Values of functions: Closures

- “Body” expression not evaluated until application
  - but type-checking takes place at compile time
  - i.e. when function is defined
- Function value =
  - `<code + environment at definition>`
  - “closure”

```ocaml
# 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
```

Binding used to eval (f ...)

<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Binding for subsequent x
Values of function application

Application: fancy word for “call”

\((e_1, e_2)\)

- “apply” the argument \(e_2\) to the (function) \(e_1\)

Application Value:
1. Evaluate \(e_1\) in current env to get (function) \(v_1\)
   - \(v_1\) is code + env
   - code is \((\text{formal } x + \text{body } e)\), env is \(E\)
2. Evaluate \(e_2\) in current env to get (argument) \(v_2\)
3. Evaluate body \(e\) in env \(E\) extended by binding \(x\) to \(v_2\)
Example 1

```
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```
Example 1

```plaintext
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```

![Diagram of variables and function evaluation]

Eval body in this env.
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;

let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
let x = 1;;
let f y =
    let x = 2 in
    fun z -> x + y + z
;;

let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
Example 3

```plaintext
let f g =
  let x = 0 in
  g 2
;;

let x = 100;;

let h y = x + y;;

f h;;
```
Static/Lexical Scoping

• For each occurrence of a variable,
  - Unique place in program text where variable 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
Alternative: dynamic scoping

```ocaml
let x = 100

let f y = x + y

let g x = f 0

let z = g 0

(* value of z? *)
```