Variables and Bindings

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

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

“Bind the value of expression e to the variable x”

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”

ML begins in a “top-level” environment
- Some names bound

```
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)

“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

```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]

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

```
Example

<table>
<thead>
<tr>
<th># let x = 2+2;;</th>
<th>val x : int = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td># let y = x * x * x;;</td>
<td>val y : int = 64</td>
</tr>
<tr>
<td># let z = [x;y;x+y];;</td>
<td>val z : int list = [4;64;68]</td>
</tr>
<tr>
<td># let x = x + x ;;</td>
<td>val x : int = 8</td>
</tr>
</tbody>
</table>
```

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

| # 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 |
```

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?
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 $\rightarrow$ Values $\rightarrow$ Types

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 occurences of `fname` inside `e`

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

Occurences of `fname` inside `e` bound to “this” definition
```
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```

Functions

Type

```
$\mathcal{f} : T_1 \rightarrow T_2$
```

F takes a value of type $T_1$ and returns a value of type $T_2$

Functions

Values

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"

Example 1

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

Example 2

```plaintext
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);;
```

Values of function application

Application: fancy word for "call"

\[(\text{e1 e2})\]

- "apply" the argument e2 to the (function) e1

Application Value:
1. Evaluate e1 in current env to get (function) v1
   - v1 is code + env
   - code is (formal x + body e), env is E
2. Evaluate e2 in current env to get (argument) v2
3. Evaluate body e in env E extended by binding x to v2

Example 1

```plaintext
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;
let x = 100;;
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

```ocaml
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? *)
```