CSE 130 : Fall 2005
Programming Languages

Lecture 5:
Environments, Bindings and Scope

Ranjit Jhala
UC San Diego

Variables and Bindings

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

```
val x = 2+2;
val x = 4 : int
val y = x * x * x;
val y = 64 : int
val z = [x,y,x+y];
val z = [4,64,68] : int list
```

“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

How ML deals with variables:

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

Environments and Evaluation

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

```
val 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

- `val x = 2+2;
val x = 4 : int`

- `val y = x * x * x;
val y = 64 : int`

- `val z = [x,y,x+y];
val z = [4,64,68] : int list`

- `val x = x + x;
val x = 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”? 

- `val x = 2+2;
val x = 4 : int`

- `val f = fn y => x + y;
val f = fn : int -> int`

- `val x = x + x;
val x = 8 : int;`

- `f 0;
val it = 4 : int`

New binding: 
- No change or mutation
- Old binding frozen in µ

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 (µ ...)

Q: Why is this a good thing?

- `val x = 2+2;
val x = 4 : int`

- `val f = fn y => x + y;
val f = fn : int -> int`

- `val x = x + x;
val x = 8 ; int;`

- `f 0;
val it = 4 : int`

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 (µ ...)
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 to efficiently
  implement these “no-sharing” semantics
- Your job is to use the simplified no-sharing semantics to
  write correct, cleaner, readable, extendable systems

Function bindings
Functions are values, can bind using val

```
val fname = fn 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

```
fun fname x = e ;
```

Occurences of fname inside e bound to “this” definition

```
fun 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
  val x = e1
in
  e2
end
```

Let-in-end is an expression!
Evaluating let-in-end in env E:
1. Evaluate expr e1 in env E to get value v : t
2. Use extended E [x -> v : t] (only) to evaluate e2

```
let
  val x = 10
in
  x * x
end
```

```
let
  val y = 20
in
  x * y
end
```

Nested bindings
Evaluating let-in-end 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

```
let
  val x = 10
in
  let
    val y = 20
  in
    x * y
  end
end
```
Nested bindings

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

Example

```ocaml
fun filter (f, l) = 
  if (null l) then []
  else
    let
      val h = hd l
      val t = filter (f, tl l)
    in
      if (f h) then h::t else t
end;
```

Nested function bindings

```ocaml
val a = 20;
fun f x = 
  let
    val y = 10
    fun g z = y + z
  in
    a + (g x)
  end;
  g 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-end` expression
    - The `let`-binding is visible (in scope) inside `in`-expression
    - Elsewhere the binding is not visible

Static Scoping

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

- Static: Can be 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