

CSE 130: Fall 2011

Programming Languages

Lecture 4: Variables and Scope

Ranjit Jhala
UC San Diego

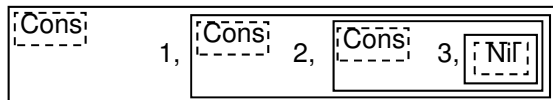


What about “Recursive” types ?

```
type int_list =  
  Nil  
| Cons of int * int_list
```

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`



News

- PA 1 due tomorrow at 5pm
- Lets first wrap up with data types...

Lists aren't built-in !

```
datatype int_list =  
  Nil  
| Cons of int * int_list
```

Lists are a **derived** type: built using elegant core!

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

`::` is just a pretty way to say “Cons”

`[]` is just a pretty way to say “Nil”

Some functions on Lists : Length

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

Base pattern

Base Expression

Ind pattern

Inductive Expression

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

Matches everything, no binding

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

Pattern-matching in order
- Must match with Nil

6

Code Mirrors Data

Some functions on Lists : listMax

```
let rec listMax xs =  
  match xs with  
  | Nil -> 0  
  | Cons(h,t) -> max h (listMax t)
```

Base pattern

Base Expression

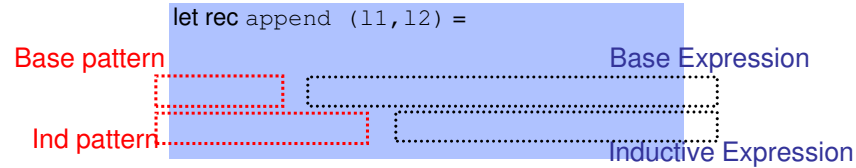
Ind pattern

Inductive Expression

Code Mirrors Data

6

Some functions on Lists : Append



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

Well designed datatype gives strategy

10

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!

Code Mirrors Data

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:

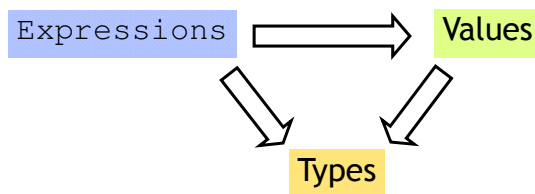
- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) * (3.78 - 5.92) = -14.766$

Q: Whats a ML datatype for such expressions ?

Code Mirrors Data



Next: Variables



Variables and Bindings

Q: How to use variables in ML ?

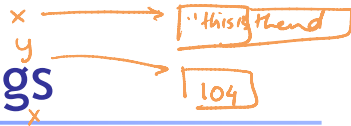
Q: How to “assign” to a variable ?

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

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

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

Variables and Bindings



```

# 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 and Evaluation

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)

Environments (“Phone Book”)

How ML deals with variables

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

ENVIRONMENT

10 Prospect Vw,	P
22 Sheff Moor Rd	PJ
5 Arnold Royd, B	R
1041 Mancheste	R
9 St Pauls Gro, B	R
10 Varley Rd, Sla	R
156 Wilson Rd, V	R
1 Wood St, Sla	Robert
2 Cheriton Dv, Q	RA
5 Dirker Dv, Mar	RA
Dirker Bank Cott,	RB
16 Holts La, Clay	RC
46 Stones Lane, I	RD
37 Laburnum Gr	RW
160 Bacup Rd, T	S
35 Markfield Av,	S
9 Brambling Dv,	SP
22b Albert Vw, Pi	T
13 Industrial Rd,	T
39 Whitley Av, Be	TE
17 Gregory Ct, Cl	V
43 Bolehill Pk, Bri	W

:	:
x	4 : int
y	64 : int
z	[4;64;68] : int list
x	8 : int

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!

⋮	⋮
x	4 : int

⋮	⋮
x	4 : int
y	64 : int

⋮	⋮
x	4 : int
y	64 : int
z	[4;64;68] : int list

⋮	⋮
x	4 : int
y	64 : int
z	[4;64;68] : int list
x	8 : int

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" ?

```
# 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
```

⋮	⋮
x	4 : int

⋮	⋮
x	4 : int
f	fn <code, ⬆ >: int->int

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" ?

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

⋮	⋮
x	4 : int

⋮	⋮
x	4 : int
f	fn <code, ⬆ >: int->int

⋮	⋮
x	4 : int
f	fn <code, ⬆ >: int->int
x	8 : int

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" ?

```
# 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 ...)

⋮	⋮
x	4 : int
f	fn <code, ⬆ >: int->int
x	8 : int

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 ?

```
# 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 ...)

⋮	⋮
x	4 : int
f	fn <code> ↑ >: int->int
x	8 : int

Binding for subsequent x

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

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

Function bindings

Functions are values, can bind using `val`

```
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)
```

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-in is an expression!

```
let x = e1 in
  e2
;;
```

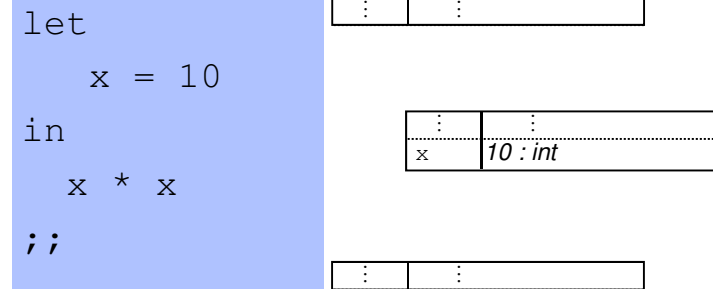
Evaluating let-in in env E :

1. Evaluate expr $e1$ in env E to get value $v:t$
2. Use extended $E[x \mapsto v:t]$ (only) to evaluate $e2$

Local bindings

Evaluating let-in in env E :

1. Evaluate expr $e1$ in env E to get value $v:t$
2. Use extended $E[x \mapsto v:t]$ to evaluate $e2$

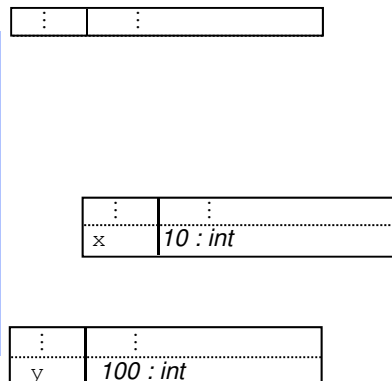


Let-in is an expression!

Evaluating let-in in env E :

1. Evaluate expr $e1$ in env E to get value $v:t$
2. Use extended $E[x \mapsto v:t]$ to evaluate $e2$

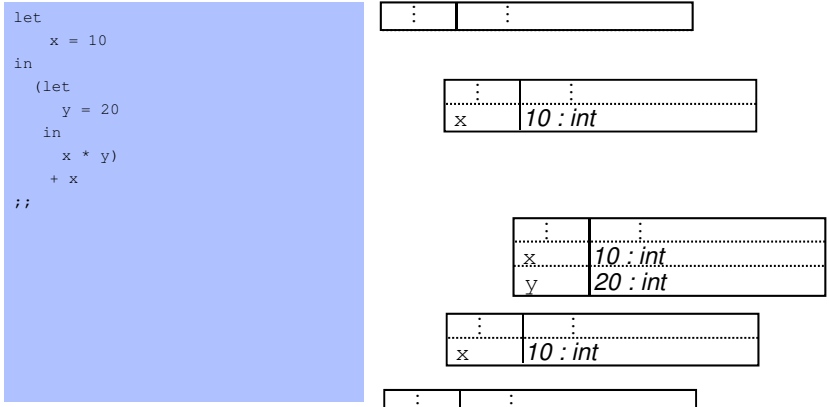
```
let y =
  let
    x = 10
  in
    x * x
;;
```



Nested bindings

Evaluating let-in in env E :

1. Evaluate expr $e1$ in env E to get value $v:t$
2. Use extended $E[x \mapsto v:t]$ to evaluate $e2$



Nested bindings

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

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

Correct Formatting

Example

```
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

```
let a = 20;;

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

f 0;
```

Env frozen with function

- Used to evaluate fun application
- Values in application are those frozen in env at definition

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

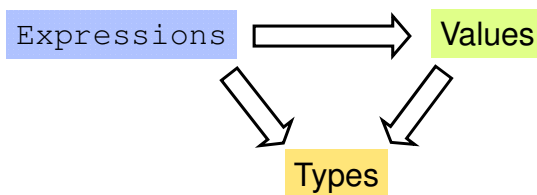
Recap

- 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** ?