

CSE 130 : Spring 2011

Programming Languages

Lecture 3:
Crash Course Ctd,
Expressions and Types

Ranjit Jhala
UC San Diego



A shorthand for function binding

```
# 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

If arg “matches”
this pattern... ..then use
this Body Expr

```
- 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 is5gte list1;;  
val it : int list = [1;4;2]  
# filter even list1;;  
val it : int list = [12;4;2;10]
```

Put it together: a “partition” function

```
# 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;;  
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

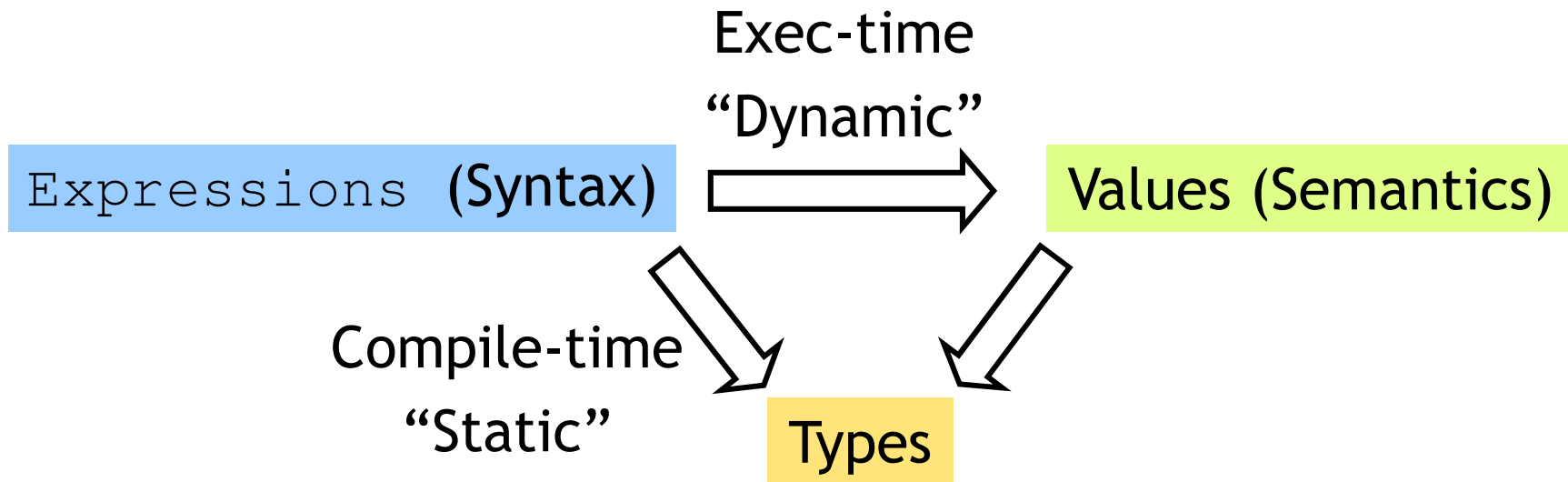
```
let rec sort xs =  
  match xs with  
  | [] -> []  
  | (h::t) ->  
    let (l,r) = partition ((<) h) t in  
    (sort l)@(h::(sort r))  
  ;;
```

Now, lets begin at the beginning ...

News

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

Begin at the beginning ...



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

2;	2	i	$i : \text{int}$	$i \Rightarrow i$
2+3;	5	$e1 + e2$	$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$	$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 + e2 \Rightarrow v1 + v2}$
7-4;	3	$e1 - e2$	$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 - e_2 : \text{int}}$	$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 - e2 \Rightarrow v1 - v2}$
(2+3) * (7-4);	15	$e1 * e2$	$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 * e_2 : \text{int}}$	$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 * e2 \Rightarrow v1 * v2}$

Expressions built from sub-expressions

Types computed from types of sub-expressions

Values computed from values of sub-expressions

Base Type: bool

true

true

b

b:bool

b => b

not (2<3)

false

not e

$$\frac{e : \text{bool}}{\text{not } e : \text{bool}}$$

$$\frac{e \Rightarrow v}{\text{not } e \Rightarrow \text{not } v}$$

not (2<3)
 &&
("ab"="cd")

false

e1 && e2

$$\frac{e_1:\text{bool} \ e_2:\text{bool}}{e_1 \&\& e_2:\text{bool}}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \&\& e_2 \Rightarrow v_1 \&\& v_2}$$

Base Type: bool

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

("ab"="cd")

false

$e1 = e2$

$\frac{e_1:T \quad e_2:T}{e_1=e_2: bool}$

$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1=e2 \Rightarrow v1=v2}$

[1;2] < [3]

false

$e1 < e2$

$\frac{e_1:T \quad e_2:T}{e_1 < e_2: bool}$

$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2}{e1 < e2 \Rightarrow v1 < v2}$

Type Errors

"pq" ^ 9 ;

$$\frac{e_1 : \text{string} \quad e_2 : \text{string}}{e_1 \wedge e_2 : \text{string}}$$

(2 + "a") ;

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

- 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

0 * (2 + "a") ;

Complex types: Tuples

- Can be of any fixed size

`(9-3, "ab" ^ "cd", 7>8)`

`(6, "abcd", false)`

`(int * string * bool)`

$$\frac{e1:T1 \quad e2:T2 \quad \dots \quad en:Tn}{(e1, e2, \dots, en) : T1 * T2 * \dots * Tn}$$

$$\frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad \dots \quad en \Rightarrow vn}{(e1, e2, \dots, en) \Rightarrow (v1, v2, \dots, vn)}$$

- Elements can have different types
- Tuples can be nested in other tuples

Complex types: Records

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

```
{ name : string, age : int, pass : bool }
```

Records are tuples with named elements...

```
{name="ranjit";age=31;pass=false}.age
```

31

int

```
{age=31,name="ranjit";pass=false}.age
```

31

int

```
{age=31;name="ranjit";pass=false}.pass
```

false

bool

Lists: Construct

Nil operator

`[]`

`[] : 'a list`

`[] => []`

Cons operator

`1 :: [2; 3]`

`[1; 2; 3]`

`int list`

$$\frac{e1:T \quad e2:T \text{ list}}{e1::e2 : T \text{ list}}$$
$$\frac{e1=>v1 \quad e2=>v2}{e1::e2 => v1::v2}$$

Lists: Deconstruct

Head

$$\frac{e : T \text{ list}}{hd\ e : T}$$

$$\frac{e \Rightarrow v1::v2}{hd\ e \Rightarrow v1}$$

Tail

$$\frac{e : T \text{ list}}{tl\ e : T \text{ list}}$$

$$\frac{e \Rightarrow v1::v2}{tl\ e \Rightarrow v2}$$

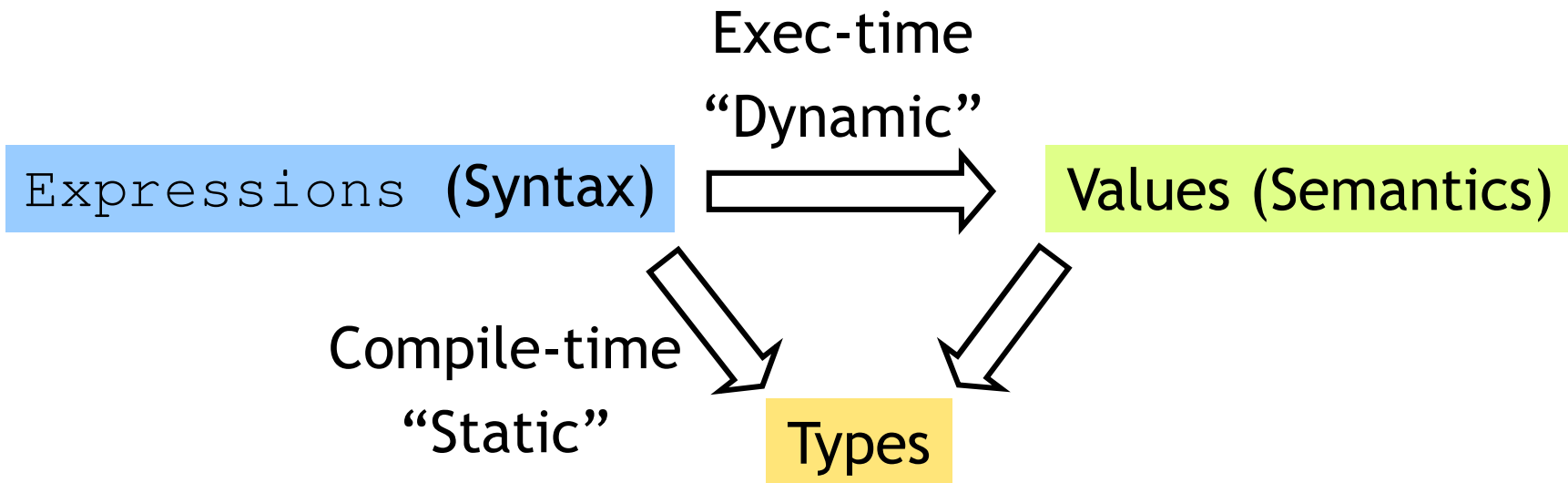
(hd [[]]; [1; 2; 3]) = (hd [[]]; ["a"])

int list

$$\frac{\underline{e_1:T} \quad \underline{e_2:T}}{e_1=e_2 : bool}$$

string list

Recap



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

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

```
if (1 < 2) then 5 else 10
```

5

int

```
if (1 < 2) then ["ab", "cd"] else ["x"]
```

["ab", "cd"]

string list

If-then-else is also an expression!

Can use any expression in then, else branch

```
if e1 then e2 else e3
```

$$\frac{e1 : \text{bool} \quad e2 : T \quad e3 : T}{\text{if } e1 \text{ then } e2 \text{ else } e3 : T}$$
$$\frac{e1 \Rightarrow \text{true} \quad e2 \Rightarrow v2}{\text{if } e1 \text{ then } e2 \text{ else } e3 \Rightarrow v2}$$
$$\frac{e1 \Rightarrow \text{false} \quad e2 \Rightarrow v3}{\text{if } e1 \text{ then } e2 \text{ else } e3 \Rightarrow v3}$$

If-then-else expressions

```
if (1 < 2) then [1;2] else 5
```

```
if false then [1;2] else 5
```

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

$e1$	:	$bool$		$e2$:	T		$e3$:	T
$if\ e1\ then\ e2\ else\ e3\ :T$								

If-then-else expressions

$$\frac{e1 : \text{bool} \quad e2 : T \quad e3 : T}{\text{if } e1 \text{ then } e2 \text{ else } e3 : T}$$

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

```
if 1>2 then [1,2] else [] []
```

int list

```
if 1<2 then [] else ["a"] []
```

string list

```
(if 1>2 then [1,2] else [])=(if 1<2 then [] else ["a"])
```

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 (“Phone Book”)

How ML deals with variables

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

01274 881373
w, Queensbury 01274 881373
Road, Bradford 01274 603920
l, Brighouse 01484 722933
ster Rd, Linthwaite 01484 844586
o, BD6 01274 679404
Slaithwaite 01484 843163
d, Wyke 01274 675753
Slaithwaite 01484 843681
Queensbury 01274 818683
larsden 01484 844450
ott, Plains, Marsden 01484 844996
layton 01274 816057
ie, Linthwaite 01484 846885
Gro, Cross Roads 01535 643681
l, Todmorden 01706 818413
Av, Bradford 01274 672644
Dv, Queensbury 01274 818887
v, Pellon 01422 259543
Rd, Sowerby Bdge 01422 839907
v, Beechwood 01422 831577
t, Clayton 01274 882408
c, Brighouse 01484 714532

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

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

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”

- 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
```

⋮	⋮
---	---

⋮	⋮
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

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” ?

```
# 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, 

⋮	⋮
x	4 : int
f	fn <code, 
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 (**£** ...)

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 (**£** ...)

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

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

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 e_1 in env E to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ to evaluate e_2

```
let
```

```
    x = 10
```

```
in
```

```
    x * x
```

```
;;
```

⋮	⋮
---	---

⋮	⋮
x	10 : int

⋮	⋮
---	---

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  
;;
```

⋮	⋮
---	---

⋮	⋮
x	10 : int

⋮	⋮
y	100 : int

Nested bindings

Evaluating let-in in env E :

1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ to evaluate e_2

```
let
  x = 10
in
  (let
    y = 20
  in
    x * y)
+ x
;;
```

⋮	⋮
---	---

⋮	⋮
x	10 : int

⋮	⋮
x	10 : int
y	20 : int

⋮	⋮
x	10 : int

⋮	⋮
---	---

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

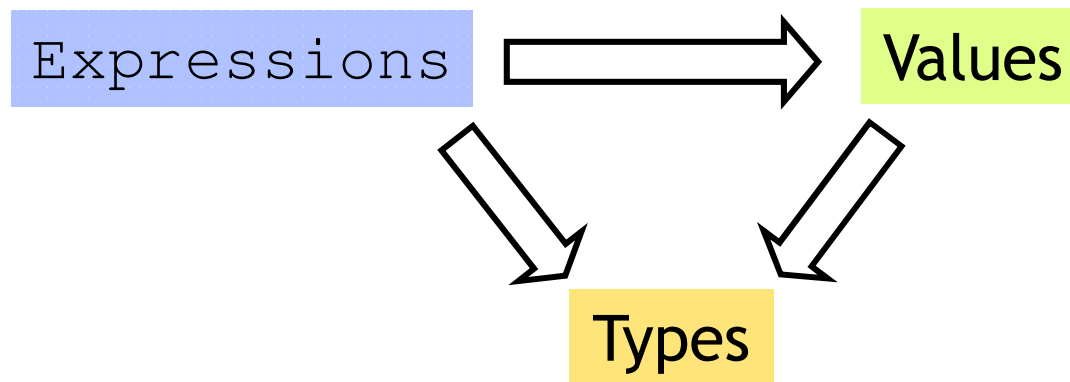
Recap

- 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** ?