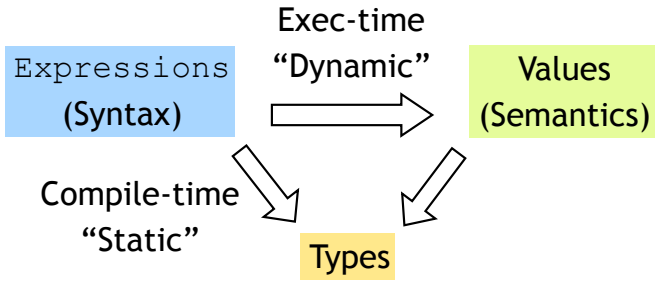
OCaml

The PL for the discerning hacker.

ML Flow



1. Enter expression
2. ML infers a type τ
3. ML crunches expression down to a value
4. Value guaranteed to have type τ

Typing -> Eval Always Works

Complex types: Lists

`[];`	[]	'a list
`[1;2;3];`	[1;2;3]	int list
`[1+1;2+2;3+3;4+4];`	[2;4;6;8]	int list
`["a";"b"; "c"^"d"];`	["a";"b"; "cd"]	string list
`[(1,"a"^"b");(3+4,"c")];`	[(1,"ab");(7,"c")]	(int*string) list
`[[1];[2;3];[4;5;6]];`	[[1];[2;3];[4;5;6]];	(int list) list

- Unbounded size
- Can have lists of anything (e.g. lists of lists)
- But…

Complex types: Lists

```
[1; "pq"];
```

All elements must have same type

Question 1

Which of these causes a type error?

(a) [1; 2; 3]
(b) [“1”, 2, 3]
(c) “[1; 2; 3]”
(d) (1, 2, 3)
(e) [“1”; 2; 3]

Complex types: Lists

List operator “Cons” ::

`1::[];`	[1]	int list
`1::[2;3];`	[1;2;3]	int list
`“a”::[“b”;“c”];`	[“a”;“b”;“c”]	string list

Can only “cons” element to a list of same type

```
1::[“b”; “cd”];
```


Lists: Construct

Nil operator

`[]` `[] : 'a list` `[] => []`

Cons operator

`1::[2;3]` int list [1;2;3]

$$\frac{e1 : T \quad e2 : T\ list}{e1::e2 : T\ list}$$

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

Complex types: Lists

List operator “Append” @

`[1;2]@[3;4;5];`	[1;2;3;4;5]	int list
`[“a”]@[“b”];`	[“a”;“b”]	string list
`[]@[1];`	[1]	int list

Can only append two lists `1 @ [2;3];`

… of the same type `[1] @ [“a”;“b”];`

Complex types: Lists

List operator “head” hd

`hd [1;2];`	1	int
`hd (["a"]@["b"]);`	“a”	string

Only take the head a nonempty list `hd [];`

Complex types: Lists

List operator “tail” tl

`tl [1;2;3];`	[2;3]	int list
`tl (["a"]@["b"]);`	[“b”]	string list

Only take the tail of nonempty list `tl [];`

Question 2: What is result of?

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

(a) Syntax Error
(b) true : bool
(c) false : bool
(d) Type Error (hd)
(e) Type Error (=)

Lists: Deconstruct

Head

$$\frac{e : T\ list}{hd\ e : T} \qquad \frac{e \Rightarrow v1::v2}{hd\ e \Rightarrow v1}$$

Tail

$$\frac{e : T\ list}{tl\ e : T\ list} \qquad \frac{e \Rightarrow v1::v2}{tl\ e \Rightarrow v2}$$

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

int list — string list

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

Recap: Tuples vs. Lists ?

What's the difference ?

- Tuples:
  - Different types, but fixed number:
    - (3, "abcd") (int * string)
    - pair = 2 elts
    - (3, "abcd",(3.5,4.2)) (int * string * (float * float))
    - triple = 3 elts
- Lists:
  - Same type, unbounded number:
    - [3;4;5;6;7] int list
- Syntax:
  - Tuples = comma Lists = semicolon

So far, a fancy calculator…

… what do we need next ?

So far, a fancy calculator…

Branches

Question 3: What is result of?

```
if (1 < 2) then true else false
```

(a) Syntax Error
(b) `true`
(c) `false`
(d) Type Error

Question 4: What is result of?

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

(a) Syntax Error
(b) [1;2]
(c) 5
(d) Type Error

If-then-else expressions

$$\frac{e1 : bool \qquad e2 : T \qquad e3 : T}{\textbf{if } e1 \textbf{ then } e2 \textbf{ else } e3 : T}$$

- 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"])
```


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

$$\frac{e1 : bool \qquad e2 : T \qquad e3 : T}{\textbf{if } e1 \textbf{ then } e2 \textbf{ else } e3 : T}$$

So far, a fancy calculator…

Variables

Question 5: I got this @ prompt

```
# [x+x; x*x] ;;
- : int list = [20; 100]
```

What had I typed before?

(a) `x = 10;`

(b) `int x = 10;`

(c) `x == 10;`

(d) `let x = 10;`

(e) `x := 10;`

Variables and bindings

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

"Bind the value of expression e to the variable x"

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


Variables and bindings

Later declared expressions can use x

- Most recent "bound" value used for evaluation

```
# 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]
#
```


Variables and bindings

Undeclared variables
(i.e. without a value binding)
are not accepted !

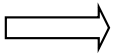
```
# let p = a + 1;
Characters 8-9:
  let p = a + 1 ;;
          ^
Unbound value a
```

Catches many bugs due to typos

Local bindings

… for expressions using “temporary” variables

```
let
  tempVar = x + 2 * y
in
  tempVar * tempVar
;;
```



17424 int

- `tempVar` is bound only inside expr body from `in` …
- Not visible (“not in scope”) outside

Question 6: What is result of?

```
let x = 10 in
(let z = 10 in x + z) + z
```

(a) Syntax Error
(b) 30
(c) Unbound Error -- x
(d) Unbound Error -- z
(e) Type Error

Binding by Pattern-Matching

Simultaneously bind several variables

```
# let (x,y,z) = (2+3,"a"^"b", 1::[2]);;
val x : int = 5
val y : string = "ab"
val z : int list = [1;2]
```


Binding by Pattern-Matching

But what of:

```
# let h::t = [1;2;3];;
Warning P: this pattern-matching not exhaustive.
val h : int = 1
val t : int list = [2;3]
```

Why is it whining ?

```
# let h::t = [];
Exception: Match_failure
# let xs = [1;2;3];
val xs = [1;2;3]: list
- val h::t = xs;
Warning: Binding not exhaustive
val h = 1 : int
val t = [2;3] : int
```

In general `xs` may be empty (match failure!)

Another useful early warning

Binding by Pattern-Matching

But what of:

NEVER USE PATTERN MATCHING LIKE THIS

```
let h::t = ...
```

ALWAYS USE THIS FORM INSTEAD

```
match l with ...
```

(coming up soon, but this is important)

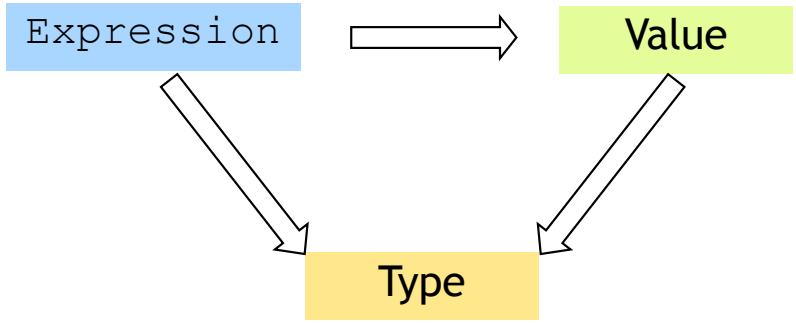
Wh

In

Another useful early warning

Functions

Functions up now, remember …



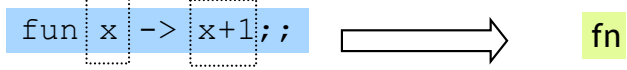
Everything is an expression
Everything has a value
Everything has a type

A function is a value!

Complex types: Functions!

Parameter (formal)

Body Expr

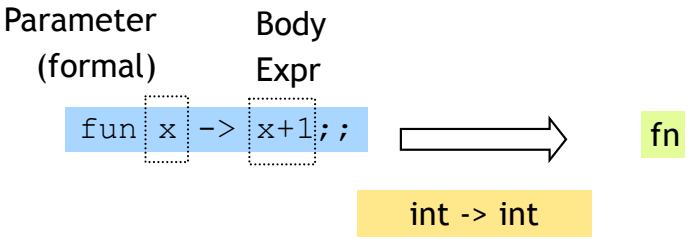


fn

int -> int

```
# let inc = fun x -> x+1 ;
val inc : int -> int = fn
# inc 0;
val it : int = 1
# inc 10;
val it : int = 11
```

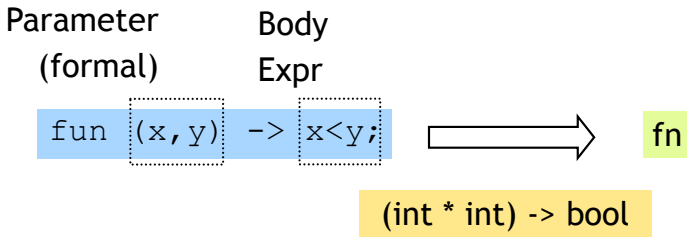
A Problem



Functions only have ONE parameter ?!

How a call ("application") is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate "Body expr"

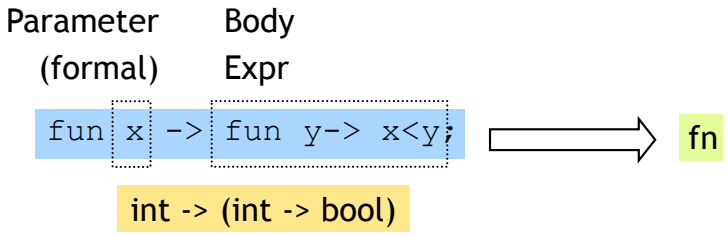
A Solution: Simultaneous Binding



Functions only have ONE parameter ?

How a call ("application") is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate "Body expr"

Another Solution ("Currying")



Whoa! A function can return a function

```
# let lt = fun x -> fun y -> x < y ;
val lt : int -> int -> bool = fn
# let is5Lt = lt 5;
val is5lt : int -> bool = fn;
# is5lt 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
```


Question 7: What is result of?

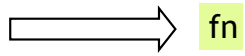
```
(fun x -> not x)
```

(a) Syntax Error
(b) <fun> : **int -> int**
(c) <fun> : **int -> bool**
(d) <fun> : **bool -> int**
(e) <fun> : **bool -> bool**

And how about…

Parameter (formal) — Body Expr

```
fun f -> fun x -> not(f x);
```



fn

('a ->bool) -> ('a -> bool)

A function can also take a function argument

```
# let neg = fun f -> fun x -> not (f x);
val lt : 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;
(*…odd, even …*)
```


Question 8: What is result of?

```
(fun f -> (fun x -> (f x)+x))
```

(a) Syntax Error

(b) Type Error

(c) <fun> : **int -> int -> int**

(d) <fun> : **int -> int**

(e) <fun> : **(int->int)-> int -> int**

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 xs =
    match xs with
    | []       -> []
    | (x::xs')-> if f x
                 then x::(filter f xs')
                 else (filter f xs');;
val filter : ('a->bool)->'a list->'a lisi) = 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 …