Hello! My name is Zach.

I’ll be your guide for today.

ML Flow

1. Enter expression
2. ML infers a type $\tau$
3. ML crunches expression down to a value
4. Value guaranteed to have type $\tau$
Complex types: Lists

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

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]
Lists: Construct

Nil operator

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

Cons operator

\[
1 :: [2;3] \\
[1;2;3]
\]

\[
e1 : T \\
e2 : T list \\
e1 :: e2 : T list \\
e1 => v1 \\
e2 => v2 \\
e1 :: e2 => v1 :: v2
\]

Complex types: Lists

List operator “Append” @

\[
[1;2]@[3;4;5]; \\
[1;2;3;4;5] \\
\]

\[
["a"]@["b"] ; \\
["a";"b"]
\]

\[
[]@[1] ; \\
[1]
\]

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

\[
\]

\[
[2;3] \\
\]

\[
["b"]
\]

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 (=)

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?

\[
\text{if } (1 < 2) \text{ then } true \text{ else } false
\]

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

Question 4: What is result of?

\[
\text{if } (1 < 2) \text{ then } [1;2] \text{ else } 5
\]

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

If-then-else expressions

\[
e_1 : bool \quad e_2 : T \quad e_3 : T
\]

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\]

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

\[
\text{if } 1>2 \text{ then } [1,2] \text{ else } [] \quad \text{if } 1<2 \text{ then } [] \text{ else } ["a"]
\]

(int list)

(string list)

\[
(\text{if } 1>2 \text{ then } [1,2] \text{ else } []) = (\text{if } 1<2 \text{ then } [] \text{ else } ["a"])
\]
If-then-else expressions

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

Variables

Question 5: I got this @ prompt

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

What had I typed before?

(a) \( x = 10 \);
(b) \( \text{int } x = 10 \);
(c) \( x == 10 \);
(d) \( \text{let } x = 10 \);
(e) \( x := 10 \);

Variables and bindings

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

“Bind the value of expression \( e \) to the variable \( x \)”
Variables and bindings
Later declared expressions can use \texttt{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
;;
```

\texttt{tempVar} is bound only inside \texttt{expr body}
from \texttt{in} \ldots
\texttt{Not visible} (“not in scope”) outside

Question 6: What is result of?
\texttt{let x = 10 in}
\texttt{(let z = 10 in x + z) + z}

\texttt{(a) Syntax Error}
\texttt{(b) 30}
\texttt{(c) Unbound Error -- x}
\texttt{(d) Unbound Error -- z}
\texttt{(e) Type Error}
Binding by Pattern-Matching

Simultaneously bind several variables

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

Why is it whining?

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

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

Another useful early warning

NEVER USE PATTERN MATCHING LIKE THIS

ALWAYS USE THIS FORM INSTEAD

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

match l with ...

(coming up soon, but this is important)

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
fun x -> x+1;;
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 ?!

Parameter (formal)  Body Expr
fun x -> x+1;;
fn
int -> int

A Solution: Simultaneous Binding

Parameter (formal)  Body Expr
fun (x,y) -> x<y;
fn
(int * int) -> bool

Functions only have ONE parameter ?!

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>(formal)</td>
<td>Expr</td>
</tr>
<tr>
<td>fun x -&gt;</td>
<td>fun y -&gt; x&lt;y;</td>
</tr>
<tr>
<td>int -&gt; (int -&gt; bool)</td>
<td></td>
</tr>
</tbody>
</table>

Whoa! A function can return a function

```ocaml
define lt (x y) = x < y

let is5lt = lt 5

let is5gte = not is5lt

is5gte 10

is5gte 2
```

And how about...

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>(formal)</td>
<td>Expr</td>
</tr>
<tr>
<td>fun f -&gt;</td>
<td>fun x -&gt; not (f x);</td>
</tr>
<tr>
<td>('a -&gt; bool) -&gt; ('a -&gt; bool)</td>
<td></td>
</tr>
</tbody>
</table>

A function can also take a function argument

```ocaml
# let neg = fun f x -> not (f x);

val lt : int -> int -> bool = fn
# let is5gte = neg is5lt;

let is5gte 10;

let it : bool = false;

(*...odd, even ...*)
```

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

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

```ocaml
# 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

```ocaml
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 list = fn

# let list1 = [1;31;12;4;7;2;10];
# filter is5lt list1 ;
val it : int list = [31;12;7;10]
# filter even list1 ;
val it : int list = [12;4;2;10]
```

Put it together: a “partition” function

```ocaml
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 …

```ocaml
# 2 <= 3;; ...
val it : bool = true
# "ba" <= "ab";;
val it : bool = false

# let lt = (<) ;;
val it : 'a -> 'a -> bool = fn

# let ltl = lt 5 ;
val it : bool = true;
# ltl 2 3;;
val it : bool = false;
```

```ocaml
# 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, let’s begin at the beginning ...