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
Side note: Tail Recursion

- Tail recursion: for each recursive call, the value of the recursive call is immediately returned.

```ocaml
let fact n =  
  if n <= 0 then 1  
  else n * fact (n - 1)
```
Side note: Tail Recursion

let rec fact n =  
    if n <= 0 then 1  
    else n * fact (n - 1)

let rec fact_tr n res =  
    if n <= 0 then res  
    else fact_tr (n - 1) (n*res)

let fact n = fact_tr n 1
Base Types

\[ \begin{align*}
5 & \rightarrow 5 & 6 & \rightarrow 6 \\
\overline{5 + 6} & \rightarrow 11 & \overline{7} & \rightarrow 7 \\
(5 + 6) + 7 & \rightarrow 18 \\
5 & \rightarrow 5
\end{align*} \]
Base Type: int

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions
Base Type: int

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions
Base Type: float

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions
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 \Rightarrow b

- 2 < 3
- true
- e1 < e2
- e1 < e2 : bool
- e1 \Rightarrow v1 \quad e2 \Rightarrow v2
- e1 < e2 \Rightarrow v1 < v2

- not(2<3)
- false
- not e
- e : bool
- not e : bool
- not e \Rightarrow not v

- ("ab"="cd")
- false
- e1 = e2
- e1 = e2 : bool
- e1 = e2 \Rightarrow v1 = v2

- not (2<3)
- false
- e1 && e2
- e1 && e2 : bool
- e1 && e2 \Rightarrow v1 && v2

- ("ab"="cd")
Base Type: bool

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

\[
\begin{align*}
\text{"ab"} &= \text{"cd"} & \text{false} & \quad e_1 = e_2 \\
_1 : _2 : \text{bool} & \quad \Rightarrow v_1 = v_2
\end{align*}
\]
Type Errors

- 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

```
"pq" ^ 9;
```
```e_1: string e_2: string
  e_1^e_2: string
```
```
(2 + "a");
```
```e_1: int e_2: int
  e_1 + e_2: int
```
```
0 * (2 + "a");
```
Complex types: Tuples

(2+2 , 7>8);

(4,false)

int * bool
Complex types: Tuples

\[(2+2, 7>8)\]; \quad (4, \text{false})

\text{int} \ast \text{bool}

\text{e_1: } T_1 \quad \text{e_2: } T_2
\quad (e_1, e_2) : T_1 \ast T_2

\text{e_1} \Rightarrow \text{v_1} \quad \text{e_2} \Rightarrow \text{v_2}
\quad (e_1, e_2) \Rightarrow (v_1, v_2)
Complex types: Tuples

- Can be of any fixed size

\[(9-3, "ab","cd", 7>8)\] \[\text{(6, "abcd", false)}\] \[
\text{(int * string * bool)}
\]

- Elements can have different types

- Tuples can be nested in other tuples

\[e_1:T_1 \ e_2:T_2 \ldots \ e_n:T_n\]
\[(e_1, e_2, \ldots, e_n): T_1 * T_2 * \ldots * T_n\]

\[e_1 \Rightarrow v_1 \ e_2 \Rightarrow v_2 \ldots \ e_n \Rightarrow v_n\]
\[(e_1, e_2, \ldots, e_n) \Rightarrow (v_1, v_2, \ldots, v_n)\]
Complex types: Records

{name="sarah" ;
age=31;
pass=false}

Records are tuples with named elements...

{name="sarah";age=31;pass=false}.age

{age=31;name="sarah";pass=false}.age

{age=31;name="sarah";pass=false}.pass
But wait...

- All evaluation rules look like:

\[
e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2
\]

\[
e_1 \ OP \ e_2 \ \Rightarrow \ v_1 \ OP \ v_2
\]
Complex types: Lists

- Unbounded size
- Can have lists of anything (e.g. lists of lists)
Complex types: Lists
Complex types: Lists

- \[[\]
- \[[\]: 'a list
- \[[\] \Rightarrow []

\[e_1; e_2; e_3; \ldots\]

\[e_1: T\] \(\Rightarrow\) \(v_1\)
\[e_2: T\] \(\Rightarrow\) \(v_2\)
\[e_3: T\] \(\Rightarrow\) \(v_3\)

\[e_1; e_2; e_3; \ldots\]: \(T\) list

\[e_1; e_2; e_3; \ldots\] \(\Rightarrow\) \[v_1; v_2; \ldots\]

All elements have the same type

\[[1; "pq"]\]
Complex types: list ..construct
Complex types: list ..construct

Cons “operator”

\[
\begin{align*}
e_1 : T & \quad e_2 : T \text{ list} \\
e_1 :: e_2 & : T \text{ list}
\end{align*}
\]

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

\[
1 :: [\text{“b”; “cd”}] ;
\]

\[
1 :: [2; 3] \quad [1; 2; 3] \quad \text{int list}
\]
Complex types: list ...construct

Append “operator”

\[
\begin{align*}
e1 : T \text{ list} \quad & e2 : T \text{ list} \\
e1 \@ e2 : T \text{ list} \\
e1 \Rightarrow v1 \quad & e2 \Rightarrow v2 \\
e1 \@ e2 \Rightarrow v1 \@ v2
\end{align*}
\]

Can only append lists of the same type
Complex types: list ... deconstruct

Reading the elements of a list:
• Two “operators”: \texttt{hd} (head) and \texttt{tl} (tail)

- \begin{align*}
\text{hd [1;2;3;4;5]} & \quad 1 \\
\text{tl [1;2;3;4;5]} & \quad [2;3;4;5] \\
\text{hd ["a";"b";"cd"]} & \quad "a" \\
\text{tl ["a";"b";"cd"]} & \quad ["b";"cd"] \\
\text{hd [(1,"a");(7,"c")]} & \quad (1,"a") \\
\text{tl [(1,"a");(7,"c")]} & \quad [(7;"c")]
\end{align*}
List: Heads and Tails
List: Heads and Tails

Head

\[ \frac{e : T \text{ list}}{\text{hd } e : T} \quad \frac{e \Rightarrow v1::v2}{\text{hd } e \Rightarrow v1} \]

Tail

\[ \frac{e : T \text{ list}}{\text{tl } e : T \text{ list}} \quad \frac{e \Rightarrow v1::v2}{\text{tl } e \Rightarrow v2} \]

\[ (\text{hd } [[[]]; [1;2;3]]) = (\text{hd } [[[]]; ["a"]]) \]

\( e_1 : T \quad e_2 : T \]
\( e_1 = e_2 : \text{bool} \)
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: =, <, andalso, orelse, 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

If (1 < 2) then [“ab”, “cd”] else [“x”]

If-then-else is also an expression!
Can use any expression in then, else branch

if e1 then e2 else e3
If-then-else expressions

if (1 < 2) then 5 else 10

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

If-then-else is also an expression!
Can use any expression in then, else branch

if e1 then e2 else e3

\[
e1 : \text{bool} \quad e2 : T \quad e3 : T
\]

if e1 then e2 else e3 : T

\[
e1 \Rightarrow \text{true} \quad e2 \Rightarrow v2
\]

if e1 then e2 else e3 \Rightarrow v2

\[
e1 \Rightarrow \text{false} \quad e3 \Rightarrow v3
\]

if e1 then e2 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

\[
e_1: \text{bool} \quad e_2: \text{T} \quad e_3: \text{T}
\]

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

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

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

\[
\begin{align*}
\text{if 1>2 then } [1,2] \text{ else } [] & \quad \text{int list} \\
\text{if 1<2 then } [] \text{ else } ["a"] & \quad \text{string list}
\end{align*}
\]

\[(\text{if 1>2 then } [1,2] \text{ else } []) = (\text{if 1<2 then } [] \text{ else } ["a"])\]
Variables and Bindings

Q: How to use variables in ML?

Q: How to “assign” to a variable?

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

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

“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 ("Phone Book")

How ML deals with variables
• Variables = "names"
• Values = "phone number"

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>y</td>
<td>64 : int</td>
</tr>
<tr>
<td>z</td>
<td>[4;64;68] : int list</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>
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

```ocaml
define 
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!
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”?

```ocaml
# 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
```

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”? 

```plaintext
# 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
```
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”**?

```ocaml
# 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 ...)*

```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4 : int</td>
</tr>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

**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 ($f \ldots$)

Q: Why is this a good thing?

```ocaml
# 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 ($f \ldots$)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>4 : int</td>
</tr>
<tr>
<td>$f$</td>
<td>fn &lt;code, $&gt;$: int&gt;-int</td>
</tr>
<tr>
<td>$x$</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

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
Recap: Environments

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

1. Evaluate:
   Find and use most recent value of variable

2. Extend: let \( x = e \);
   Add new binding at end of “phone book”
Next: Functions

Expressions → Types → Values
Functions

Functions are values, can bind using `let`

```ml
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`

```ml
let rec fname x = e ;;
```

Occurrences of `fname` inside `e` bound to “this” definition

```ml
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```
<table>
<thead>
<tr>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
</tbody>
</table>
Functions

\[ e_1 : T_2 \rightarrow T \quad \quad e_2 : T_2 \]
\[ e_1 \ e_2 : T \]
Two questions about function values:

What is the value:

1. ... of a function?

2. ... of a function “application” (call)?
Two questions about function values:

What is the **value**:

1. ... of a function ?

2. ... of a function “application” (call) ?
Values of functions: Closures

- “Body” expression not evaluated until application
  - but type-checking takes place at compile time
  - i.e. when function is defined
- Function value =
  - <code + environment at definition>
  - “closure”

```plaintext
# 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 \( f \ldots \)

<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>fn &lt;code, &gt;: int-&gt;int</td>
</tr>
<tr>
<td>x</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

Binding for subsequent \( x \)
Values of function application

Application: fancy word for “call”

\((e_1 \ e_2)\)

- “apply” the argument \(e_2\) to the (function) \(e_1\)

Application Value:

1. Evaluate \(e_1\) in current env to get (function) \(v_1\)
   - \(v_1\) is code + env
   - code is \((\text{formal } x + \text{body } e)\), env is \(E\)

2. Evaluate \(e_2\) in current env to get (argument) \(v_2\)

3. Evaluate body \(e\) in env \(E\ extended\) by binding \(x\) to \(v_2\)
Example 1

```plaintext
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```
Example 1

```plaintext
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```

Eval body in this env
Example 2

```ml
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;

let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
```
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;

let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
Example 3

```haskell
let f g =
  let x = 0 in
  g 2
;;

let x = 100;;

let h y = x + y;;

f h;;
```
Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable 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
Alternative: dynamic scoping

```ocaml
let x = 100

let f y = x + y

let g x = f 0

let z = g 0

(* value of z? *)
```