Programmer enters expression

1. 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

2. ML evaluates expression to compute value
   - Of the same "type" found in step 2
Tail Recursion: Factorial

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

fac 3;;
Tail recursion

• Tail recursion:
  - recursion where all recursive calls are immediately followed by a return
  - in other words: not allowed to do anything between recursive call and return
Tail recursive factorial

```ocaml
let fact x =
```
let fact x =
    let rec helper x curr =
        if x <= 0
        then curr
        else helper (x - 1) (x * curr)
    in
    helper x 1;;
let fact x =
    let rec helper x curr =
        if x <= 0
            then curr
        else helper (x - 1) (x * curr)
    in
    helper x 1;;
fact 3;;
Tail recursion

• Tail recursion:
  - for each recursive call, the value of the recursive call is immediately returned
  - in other words: not allowed to do anything between recursive call and return

• Why do we care about tail recursion?
  - it turns out that tail recursion can be optimized into a simple loop
Compiler can optimize!

let fact x =  
  let rec helper x curr =  
    if x <= 0  
    then curr  
    else helper (x - 1) (x * curr)  
  in  
  helper x 1;;

fact(x) {  
  curr := 1;  
  while (1) {  
    if (x <= 0)  
      then return curr  
    else { x := x - 1; curr := (x * curr) }  
  }  
}
Tail recursion summary

- Tail recursive calls can be optimized as a jump

- Part of the language specification of some languages (ie: you can count on the compiler to optimize tail recursive calls)
Base Types
Base Type: int

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: float

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

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions
<table>
<thead>
<tr>
<th>Base Type: bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>2 &lt; 3</td>
</tr>
<tr>
<td>not (2&lt;3)</td>
</tr>
<tr>
<td>(“ab”=”cd”)</td>
</tr>
<tr>
<td>not (2&lt;3) &amp;&amp; (“ab”=”cd”)</td>
</tr>
</tbody>
</table>
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 ?

(“ab” = “cd”) → false
\[ e_1 = e_2 \]
\[ e_1 = e_2 : bool \]
\[ e_1 \Rightarrow v_1, e_2 \Rightarrow v_2 \]
\[ e_1 = e_2 \Rightarrow v_1 = v_2 \]
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
Complex types: Tuples

(2+2 , 7>8);  
(4,false)

int * bool
Complex types: Tuples

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

\(\text{int} \ast \text{bool}\)

\(\begin{align*}
e_1 &: T_1 & e_2 &: T_2 \\
(e_1, e_2) &: T_1 \ast T_2
\end{align*}\)

\(\begin{align*}
e_1 &\Rightarrow v_1 & e_2 &\Rightarrow v_2 \\
(e_1, e_2) &\Rightarrow (v_1, v_2)
\end{align*}\)
Complex types: Tuples

- Can be of any fixed size

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

- Elements can have different types

- Tuples can be nested in other tuples
Complex types: Records

{ name: "sarah", age: 31, pass: false }

Records are tuples with named elements...

{ name: "sarah", age: 31, pass: false }.age = 31

{ age: 31, name: "sarah", pass: false }.age = 31

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

• All evaluation rules look like:

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

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

Examples:

- **Int list**: 
  \[[1+1; 2+2; 3+3; 4+4]; [2; 4; 6; 8]\]

- **String list**: 
  \[\[\"a\"; \"b\"; \"c\"; \"d\"\]; \[\"a\"; \"b\"; \"cd\"\]\]

- **(int*string) list**: 
  \[\[(1; \"a\"; \"b\") ; (3+4; \"c\")\]; \[(1; \"ab\") ; (7; \"c\")\]\]

- **Int list**: 
  \[\[[1]; [2; 3]; [4; 5; 6]\]; [[1]; [2; 3]; [4; 5; 6]]\]
Complex types: Lists
Complex types: Lists

[]

[] : 'a list

[] ⇒ []

[e1; e2; e3; …]

\[
\begin{array}{c}
e_1 : T \\
e_2 : T \\
e_3 : T \\
\vdots
\end{array}
\]

[e1; e2; e3; …] : T list

\[
\begin{array}{c}
e_1 ⇒ v_1 \\
e_2 ⇒ v_2 \\
e_3 ⇒ v_3
\end{array}
\]

[e1; e2; …] ⇒ [v1; v2; …]

All elements have the same type

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

Cons “operator”

\[
\begin{array}{c}
\text{e}_1 : T \\
\text{e}_2 : \text{T list}
\end{array}
\]
\[
\text{e}_1 \text{::} \text{e}_2 : \text{T list}
\]

\[
\begin{array}{c}
\text{e}_1 \Rightarrow \text{v}_1 \\
\text{e}_2 \Rightarrow \text{v}_2
\end{array}
\]
\[
\text{e}_1 \text{::} \text{e}_2 \Rightarrow \text{v}_1 \text{::} \text{v}_2
\]

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

1::[“b”; “cd”];

Can only “cons” element to a list of same type
Complex types: list ...construct

Append “operator”

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

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

\[
\text{int list}
\]

\[
1@["b"; "cd"];
\]

\[
[1]@["b"; "cd"];
\]

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

Reading the elements of a list:

- Two “operators”: `hd` (head) and `tl` (tail)

```plaintext
[1;2;3;4;5]  # Int list

hd [1;2;3;4;5] 1  # Int

[“a”;“b”;“cd”]  # String list

hd [“a”;“b”;“cd”] “a”  # String

[(1,”a”);(7,”c”)]  # (Int * String) list

hd [(1,”a”);(7,”c”) (1,”a”)  # Int * String

tl [(1,”a”);(7,”c”) [(7; “c”)]  # (Int * String) list

[[];[1;2;3];4;5]]  # Int list list

hd [[];[1;2;3];4;5] 1  # Int list

tl [][];[1;2;3];4;5] [2;3;4;5]  # Int list list
```
List: Heads and Tails
List: Heads and Tails

Head

\[
\begin{align*}
\text{e} : T \text{ list} & \quad \text{e} \Rightarrow v1::v2 \\
\text{hd e} : T & \quad \text{hd e} \Rightarrow v1
\end{align*}
\]

Tail

\[
\begin{align*}
\text{e} : T \text{ list} & \quad \text{e} \Rightarrow v1::v2 \\
\text{tl e} : T \text{ list} & \quad \text{tl e} \Rightarrow v2
\end{align*}
\]

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

\[
\begin{align*}
\text{int list} & \\
\text{string list}
\end{align*}
\]

\[
\begin{align*}
\text{e}_1 : T & \quad \text{e}_2 : T \\
\text{e}_1 = \text{e}_2 : \text{ bool}
\end{align*}
\]
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

5

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
**If-then-else expressions**

If (1 < 2) then 5 else 10

5

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

if e1 then e2 else e3: T

e1 : bool       e2: T       e3: T

if e1 then e2 else e3 : T

if e1 then e2 else e3 ⇒ v2

if e1 ⇒ true     e2 ⇒ v2

if e1 then e2 else e3 ⇒ v2

if e1 ⇒ false   e3 ⇒ v3

if e1 then e2 else e3 ⇒ 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

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

\[
\begin{align*}
e_1 & : \text{bool} \\
e_2 & : T \\
e_3 & : T \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\end{align*}
\]

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

\[\text{int list} \quad \text{string list}\]

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

```ocaml
# 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"

<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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```ml
# 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!
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
```

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”?

```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
```
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,</td>
</tr>
<tr>
<td></td>
<td></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** \((f \ldots)\)

<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;</td>
</tr>
<tr>
<td>(x)</td>
<td>8 : int</td>
</tr>
</tbody>
</table>

**Binding for subsequent** \(x\)
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
Functions

Functions are values, can bind using `let`

```plaintext
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`

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

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

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


Functions

<table>
<thead>
<tr>
<th>Type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 : T_2 \rightarrow T$</td>
<td>$e_1 e_2 : T$</td>
</tr>
<tr>
<td>$e_2 : T_2$</td>
<td></td>
</tr>
</tbody>
</table>
Two questions about function values:

What is the value:

1. ... of a function?

2. ... of a function “application” (call)? (e₁ e₂)
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”

```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
```

<table>
<thead>
<tr>
<th>Binding used to eval (f ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

Binding for subsequent x
Values of function application

Application: fancy word for “call”

\[(e1 \ e2)\]

• “apply” the argument \(e2\) to the (function) \(e1\)

Application Value:
1. Evaluate \(e1\) in current env to get (function) \(v1\)
   - \(v1\) is code + env
   - code is \((\text{formal } x + \text{body } e)\) , env is \(E\)
2. Evaluate \(e2\) in current env to get (argument) \(v2\)
3. Evaluate body \(e\) in env \(E\) extended by binding \(x\) to \(v2\)
Example 1

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

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);;
```
Example 2

```ocaml
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

```ocaml
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? *)
```