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
Tail Recursion: Factorial

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

let fact x =

let rec helper i acc =
let i = 0 then acc
else helper (i-1) i * acc

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;;
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

then curr

else helper (x - 1) (x * curr)

in

helper x 1;;

fact(x) {
  curr := 1;
  while (1) {
    if (x <= 0) then { return curr }

    then { return curr }

    else { x := x - 1; curr := (x * curr) }

  }
}

recursion! Loop!
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
**Base Type: int**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Type</th>
<th>Values Computed From Values of Sub-expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2;</td>
<td>2</td>
<td>i: int</td>
<td>i ⇒ i</td>
</tr>
<tr>
<td>2+3;</td>
<td>5</td>
<td>e1 + e2</td>
<td>e1⇒v1 e2⇒v2 e1+e2⇒v1+v2</td>
</tr>
<tr>
<td>7−4;</td>
<td>3</td>
<td>e1 − e2</td>
<td>e1⇒v1 e2⇒v2 e1−e2⇒v1−v2</td>
</tr>
<tr>
<td>(2+3) * (7−4);</td>
<td>15</td>
<td>e1 * e2</td>
<td>e1⇒v1 e2⇒v2 e1<em>e2⇒v1</em>v2</td>
</tr>
</tbody>
</table>

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
Base Type: bool

- true
- false
- \(2 < 3\)
- \(\text{not}(2<3)\)
- \(\text{"ab"} = \text{"cd"}\)
- \(\text{e1} < \text{e2}\)
- \(\text{e1} = \text{e2}\)
- \(\text{e1} \land \text{e2}\)

### Examples

\[
\begin{align*}
\text{true} & \implies \text{true} \\
\text{false} & \implies \text{false} \\
\text{true} & \implies \text{true} \\
\text{false} & \implies \text{false} \\
\text{2 < 3} & \implies \text{true} \\
\text{not}(2<3) & \implies \text{false} \\
\text{"ab"} = \text{"cd"} & \implies \text{false} \\
\text{true} & \implies \text{true} \\
\text{b} & \implies \text{b} \\
\text{e1 < e2} & \implies \text{false} \\
\text{e1 = e2} & \implies \text{false} \\
\text{e1} \land \text{e2} & \implies \text{false} \\
\end{align*}
\]
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  e₁ = e₂

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

\[ e₁ \Rightarrow v₁ \quad e₂ \Rightarrow v₂ \]
\[ e₁ = e₂ \Rightarrow v₁ = v₂ \]
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

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

int * bool

$e_1 : T_1$ \hspace{1cm} $e_2 : T_2$

$(e_1, e_2) : T_1 * T_2$

$e_1 \Rightarrow v_1$ \hspace{1cm} $e_2 \Rightarrow v_2$

$(e_1, e_2) \Rightarrow (v_1, v_2)$
Complex types: Tuples

- Can be of any fixed size

\[(9-3, "ab"^"cd", 7>8)\] \[(6, "abcd", false)\]

\[(\text{int } \ast \text{ string } \ast \text{ bool})\]

- Elements can have different types

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

Records are tuples with named elements...

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

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

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

31  int

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

31  int

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

false  bool
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

[] => []

[e1; e2; e3; ...]

e1: T
e2: T
e3: T...

[e1; e2; e3; ...]: T list

[e1 => v1 e2 => v2 e3 => v3]

[e1; e2; ...] => [v1; v2; ...]

All elements have the same type

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

Cons “operator”

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

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

1::[“b”; “cd”];

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

Append “operator”

\[ e_1 : T \text{ list} \quad e_2 : T \text{ list} \quad e_1 \@ e_2 : T \text{ list} \]

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

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)

- [1;2;3;4;5]
  - `hd [1;2;3;4;5]` → 1
  - `tl [1;2;3;4;5]` → [2;3;4;5]

- [“a”; “b”; “cd”]
  - `hd [“a”; “b”; “cd”]` → “a”
  - `tl [“a”; “b”; “cd”]` → [“b”; “cd”]

- [(1,”a”);(7,”c”)]
  - `hd [(1,”a”);(7,”c”)` → (1,”a”)
  - `tl [(1,”a”);(7,”c”)` → [(7; “c”)]

- [[];[1;2;3];[4;5]]
  - `hd [[];[1;2;3];4;5]` → 1
  - `tl [[];[1;2;3];4;5]` → [2;3;4;5]

- [(int * string) list]
List: Heads and Tails
List: Heads and Tails

Head

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

Tail

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

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

int list

\[
\frac{e_1 : T \ e_2 : T}{e_1 = e_2 : \text{ bool}}
\]

string list
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
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"]
```

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 : bool  e2 : T  e3 : T

if e1 then e2 else e3 : T

e1 ⇒ true  e2 ⇒ v2  if e1 then e2 else e3 ⇒ v2

e1 ⇒ false  e3 ⇒ v3  if e1 then e2 else e3 ⇒ v3
If-then-else expressions

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

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

\[
\text{if false then } [1;2] \text{ else 5}
\]
If-then-else expressions

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

\[
\begin{align*}
e1 &: \texttt{bool} \\
e2 &: T \\
e3 &: T \\
\text{if } e1 \text{ then } e2 \text{ else } e3 &: T
\end{align*}
\]

if 1>2 then [1,2] else [] = if 1<2 then [] else ["a"]
Next: Variables
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”

<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

# 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”? 

```latex
# 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
define x = 2+2
val x = 4

define f = fun y -> x + y
val f : int -> int = fn

define x = x + x
val x = 8

f 0
val it = 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>
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 …)

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 …)

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

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: \texttt{let } x = e ; ;
Add new binding at end of “phone book”
Next: Functions

Expressions → Types → Values
Functions

Functions are values, can bind using `let`

```
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)
```
<table>
<thead>
<tr>
<th>Functions</th>
<th>Type</th>
</tr>
</thead>
</table>
Functions

\[
\begin{array}{c}
e_1 : T_2 \rightarrow T \\
e_2 : T_2 \\
e_1 \ e_2 : T
\end{array}
\]
Two questions about function values:

What is the value:

1. ... of a function?

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

Two questions about function values:

What is the value:

1. ... of a function ?

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

```
# 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
| --- |   |
| f   | fn <code, >: int->int |
| x   | 8 : int |

Binding used to eval (`f ...`)

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 (formal \(x\) + 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

```
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

```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 3

```plaintext
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

let x = 100

let f y = x + y

let g x = f 0

let z = g 0

(* value of z? *)