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

Expressions (Syntax) → Values (Semantics)

- Compile-time “Static”
- Exec-time “Dynamic”

Expressions

<table>
<thead>
<tr>
<th>Compile-time “Static”</th>
<th>Types</th>
</tr>
</thead>
</table>

How does it execute?

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

fac 3;;
```

Tail Recursion: Factorial

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

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

```plaintext
let fact x =
  let rec helper i acc =
    if i <= 0
    then acc
    else helper (i-1) (i * acc)
  in helper x 1
```

Tail recursive factorial

```plaintext
let fact x =
  let rec helper x curr =
    if x <= 0
    then curr
    else helper (x - 1) (x * curr)
  in
  helper x 1;;
```
How does it execute?

```ml
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!

```ml
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
### Base Type: int

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>2 + 3</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>7 - 4</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>(2+3) * (7-4)</td>
<td>int</td>
<td>15</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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>float</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>float</td>
<td>e</td>
</tr>
<tr>
<td>1.0 . 4.0</td>
<td>float</td>
<td>1.0 . 4.0</td>
</tr>
<tr>
<td>(1.0 . 4.0) / (7.0 . 8.0)</td>
<td>float</td>
<td>1.66</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: string

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ab&quot;</td>
<td>string</td>
<td>&quot;ab&quot;</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
<td>s</td>
</tr>
<tr>
<td>&quot;abcd&quot;</td>
<td>string</td>
<td>e1 * e2</td>
</tr>
<tr>
<td>&quot;cd&quot;</td>
<td>string</td>
<td>e2</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: bool

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>bool</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>bool</td>
<td>b</td>
</tr>
<tr>
<td>2 &lt; 3</td>
<td>bool</td>
<td>e1 &lt; e2</td>
</tr>
<tr>
<td>not (2&lt;3)</td>
<td>bool</td>
<td>not e</td>
</tr>
<tr>
<td>(&quot;ab&quot; = &quot;cd&quot;)</td>
<td>bool</td>
<td>e1 = e2</td>
</tr>
<tr>
<td>not (2&lt;3)</td>
<td>bool</td>
<td>not e</td>
</tr>
<tr>
<td>(&quot;ab&quot; = &quot;cd&quot;)</td>
<td>bool</td>
<td>e1 = e2</td>
</tr>
</tbody>
</table>

**Expressions built from sub-expressions**

**Types computed from types of sub-expressions**

**Values computed from values of sub-expressions**

### 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**

```java
0 * (2 + "a");
```
Complex types: Tuples

- Can be of any fixed size

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

\[(\text{int} \,* \, \text{string} \,* \, \text{bool})\]

- Elements can have different types
- Tuples can be nested in other tuples

But wait...

- All evaluation rules look like:

\[e_1 = v_1 \, \quad e_2 = v_2 \quad e_1 \, \text{OP} \, \quad e_2 \, \Rightarrow \, v_1 \, \text{OP} \, v_2\]

Complex types: Lists

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

Complex types: Lists

Complex types: List .. construct

Complex types: list .. construct

Cons “operator”

Complex types: list .. construct

Append “operator”

Complex types: list .. construct

Reading the elements of a list:

- Two “operators”: \texttt{hd} (head) and \texttt{tl} (tail)

\texttt{hd \{1;2;3|4\}}: 1

\texttt{tl \{1;2;3|4\}}: \{2;3;4\}

\texttt{hd \{“a”;“b”;“cd”\}}: “a”

\texttt{tl \{“a”;“b”;“cd”\}}: \{“b”;“cd”\}

\texttt{hd \{(1,“a”);(7,“c”)\}}: (1,“a”)

\texttt{tl \{(1,“a”);(7,“c”)\}}: \{(7,“c”)\}

\texttt{hd \{1;2;3\}}: 1

\texttt{tl \{1;2;3\}}: \{2;3\}

All elements have the same type:

\{1; “pq”\}

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

\texttt{1::\{“b”;“cd”\}};

Can only append lists of the same type:

\texttt{1@\{“b”;“cd”\}};

Can only append lists of the same type:

\texttt{1@\{1;2\}@\{3;4\}}; \texttt{1@\{1;2;3;4\}}; \texttt{1@\{1;2;3;4;5\}};
List: Heads and Tails

List: Heads and Tails

Head

\[ e : \text{list} \quad \text{hd} \ e : T \]

Tail

\[ e : \text{list} \quad \text{tl} \ e : \text{list} \]

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

Recap

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

If-then-else expressions

\[ \text{if } (1 < 2) \text{ then } 5 \text{ else } 10 \]

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

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

\[
\begin{array}{llll}
  e_1 & : & \text{bool} & \quad e_2 : T & \quad e_3 : T \\
  \text{if} & e_1 & \text{then} & e_2 & \text{else} & e_3 & : T
\end{array}
\]

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

\[
\begin{array}{llll}
  \text{if} & 1>2 & \text{then} & \text{[1;2]} & \text{else} & \text{[]} \\
  \text{if} & 1<2 & \text{then} & \text{[]} & \text{else} & \text{["a"]}
\end{array}
\]

Next: Variables

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 \( x \)”

Variables and Bindings

Late 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”
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
```

Binding used to eval `(f …)`

Binding for subsequent `x`

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”?

```
# 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`
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?
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 but pointers 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”

Functions

Functions are values, can bind using let

\[
\text{let \ } fname = \text{ fun } x \rightarrow e \ ;
\]

Problem: Can’t define recursive functions!
• $fname$ is bound after computing rhs value
• no (or “old”) binding for occurrences of $fname$ inside $e$

\[
\text{let rec } fname \ x = e \ ;
\]

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

\[
\text{let rec } fac \ x = \text{ if } x \leq 1 \text{ then } 1 \text{ else } x \times \text{fac } (x-1)
\]
Two questions about function values:

What is the value:
1. ... of a function?
2. ... of a function “application” (call)? \((e_1 \ e_2)\)

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 =
  - \(<\text{code + environment at definition}>\)
  - “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
```

```
Binding used to eval \(\{e \ ...\}\)
```

```
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 2

```plaintext
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 g = x + g;;
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? *)
```