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

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

---

### How does it execute?

```ml
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

```ml
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?

```
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) {
      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

Expression built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions

Base Type: int
Equality testing is built-in for all expr, values, types
- but compared expressions must have the same type

**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

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

- Can be of any fixed size
- Elements can have different types
- Tuples can be nested in other tuples

But wait...

- All evaluation rules look like:

But wait...

Complex types: Records

Records are tuples with named elements...

Complex types: Lists

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

[1] \[1 : 'a list\]

\[e1; e2; e3; ...\] \[e1 = v1, e2 = v2, e3 = v3\]

\[e1; e2; \ldots : T\ list\]

\[e1; e2; \ldots \Rightarrow \{v1; v2; \ldots\}\]

All elements have the same type

\[\{1; \text{“pq”}\}\]

Complex types: Lists

Complex types: list ..construct

Cons “operator”

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

\[(1;2;3)\]

\[\text{int list}\]

\[e1; e2 : T\ list\]

\[e1 = v1, e2 = v2\]

\[e1::e2 : T\ list\]

\[e1 = v1, e2 = v2\]

\[e1::e2 \Rightarrow v1::v2\]

\[1::[\text{“b”}; \text{“cd”}]\]

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

Complex types: list ..construct

Append “operator”

\[\{1;2\}@[3;4]\]

\[1;2;3;4\]

\[\text{int list}\]

\[e1 : T\ list\]

\[e2 : T\ list\]

\[e1 = v1, e2 = v2\]

\[e1::e2 \Rightarrow v1::v2\]

\[1@[\text{“b”}; \text{“cd”}]\]

\[\{1\}@[\text{“b”}; \text{“cd”}]\]

Can only append lists of the same type

Complex types: list ...deconstruct

Reading the elements of a list:

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

\[\text{hd} \{1;2;3;4;5\} : \text{int}\]

\[\text{tl} \{1;2;3;4;5\} : \{2;3;4;5\}\]

\[\text{hd} \{\text{“a”; “b”; “cd”}\} : \text{string}\]

\[\text{tl} \{\text{“a”; “b”; “cd”}\} : \{\text{“b”; “cd”}\}\]

\[\text{hd} \{(1;2;3;4;5);\{6;7;8;9;\}\} : \text{int list}\]

\[\text{tl} \{(1;2;3;4;5);\{6;7;8;9;\}\} : \{7;8;9;\}\]

\[\text{hd} \{(1;2;3;4;5);\{6;7;8;9;\}\} : \text{int list}\]

\[\text{tl} \{(1;2;3;4;5);\{6;7;8;9;\}\} : \{6;7;8;9;\}\]

\[\text{hd} \{(1;2;3;4;5);\{6;7;8;9;\}\} : \text{int list}\]

\[\text{tl} \{(1;2;3;4;5);\{6;7;8;9;\}\} : \{1;2;3;4;5\}\]
### List: Heads and Tails

**Head**

\[
\text{head } \text{list} = \text{null}
\]

**Tail**

\[
\text{tail } \text{list} = \text{null}
\]

\[
\text{if } (1 < 2) \text{ then } ["ab","cd"] \text{ else } ["x"]
\]

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

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

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

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

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

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

### 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

### Expressions (Syntax) → Values (Semantics)

- **Compile-time “Static”**
- **Eval-time “Dynamic”**

- **Types**
  - Integers: +, -, *
  - Floats: +, -, *
  - Booleans: =, <, and also, orelse, not
  - Strings: ^

- **Tuples, Records:** #
  - Fixed number of values, of different types
  - Lists: ::, @, hd, tl, null
  - Unbounded number of values, of same type

### If-then-else expressions

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

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

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

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

Next: Variables

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”

Variables and Bindings

```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]
```

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”
Environments and Evaluation

ML begins in a “top-level” environment

- Some names bound

```ml
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

```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
```

---

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”?

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

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

let $fname = fun x \rightarrow e;$

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

let rec $fname \ x = e ;$

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

let rec $fac \ x = if \ x \leq 1 \ then \ 1 \ else \ x \cdot fac \ (x-1)$
Functions

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 =
  - $<$code + environment at definition$>$
  - “closure”

```plaintext
# let x = 2+2;; val x : int = 4
# let f = fun y -> x + y;;
# let x = 2+2 ;;
val x : int = 8
# f 0;;
val it : int = 4
```

Binding used to eval ($\ell \ ...$)

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

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

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 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

```ml
let x = 100
let f y = x + y
let g x = f 0
let z = g 0 (* value of z? *)
```