**Tail Recursion**

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

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

```plaintext
let fact_tr n res =  
  if n <= 0 then res  
  else fact_ntr (n - 1) (n*res)  
let fact n = fact_ntr n 1
```

**Complex types: Lists**

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

- `[]`: 'a list
- `[e1; e2; e3;...]`: T list
- `[] = []`

All elements have the same type

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

Cons “operator”

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

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

Append “operator”

```plaintext
[1;2]@[3;4]  [1;2;3;4]
int list
```

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]     hd [1;2;3;4;5]  1     int
tl [1;2;3;4;5]  [2;3;4;5]     int list

[“a”;“b”;“cd”]   hd [“a”;“b”;“cd”]  “a”     string
tl [“a”;“b”;“cd”]  [“b”;“cd”]     string list

[(1;“a”);(7;“c”)]   hd [(1;“a”);(7;“c”)]  (1;“a”)     int * string
tl [(1;“a”);(7;“c”)]  [(7;“c”)]     (int * string) list

[[1];[2;3];4;5]     hd [[1];[2;3];4;5]  1     int
                    tl [[1];[2;3];4;5]  [2;3;4;5]     int list
```

List: Heads and Tails

Head

```plaintext
e : T list
  hd e : T
  e :: v1 :: v2
  hd e :: v1 :: v2
```

Tail

```plaintext
e : T list
tl e : T list
  e :: v1 :: v2
  tl e :: v2
```

```plaintext
(hd [[1];[2;3]]) = (hd [[“a”]])
```

Recap

Expressions (Syntax)  \rightarrow  Values (Semantics)

Compile-time

- “Static”

```
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
```

- “Dynamic”
Recap

- Integers: +, -, *
- Floats: +, -, *
- Booleans: =, <, andalso, 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 expressions

If-then-else expressions

If-then-else expressions

Next: Variables
Variables and Bindings

Q: How to use variables in ML?
Q: How to “assign” to a variable?

\[
\textbf{let } x = e ;;
\]

“Bind the value of expression \( e \) to the variable \( x \)”

Environments (“Phone Book”)

How ML deals with variables
- 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”

Environments and Evaluation

ML begins in a “top-level” environment
- Some names bound

\[
\textbf{let } x = e ;;
\]

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. Evaluate \( \text{expr} \) in current env to get value \( v : t \)
2. Extend env to bind \( x \) to \( v : t \)
(Repeat with next binding)

Example

\[
\textbf{let } x = 2 + 2;;
\]
\[
\text{val } x : \text{int} = 4
\]

\[
\textbf{let } y = x \times x \times x;;
\]
\[
\text{val } y : \text{int} = 64
\]

\[
\textbf{let } z = \{ x; y; x + y \};;
\]
\[
\text{val } z : \text{int list} = \{ 4; 64; 68 \}
\]

Later declared expressions can use \( x \)
- Most recent “bound” value used for evaluation

Sounds like C/Java?
NO!
### 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

**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
**Function bindings**

Functions are values, can bind using `val`

```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);
```

---

**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`  ➔  `Values`

- `Types`

**Q:** What’s the value of a function?

---

**Functions**

Two questions about function values:

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

---

**Values**

- “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 ...)`

```
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 for subsequent `x`
Values of function application

Application: fancy word for “call”

• “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 = 2;;
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 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

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