The interpreter consists of several functions. The entry point is called "myeval" which takes two arguments: an expression to be evaluated and an association list representing the binding environment.

If the expression is an application, myeval calls "myapply" whose arguments are a function to be applies, and a list of argument values. The function will be a lambda expression, or a primitive, one of car, cdr, cons, eq?, null?

e.g. if the expression is (f 'a 'b) then the call to myapply will be (myapply '(lambda (x y) ...)
'(a b) env)

This is call-by-value parameter passing since myapply gets passed a list of the argument values.

Scheme has a way to use local definition for a variable – temp. variables.

e.g. (let ((x 3) ← variable, ← value
(y 4))

x, y are local variables

body ⟶ (+ x y))

(let ((x (f ...)))
(* x x))

is better than (* (f ...) (f ...))

let is actually the application of a lambda expression

(let ((x 3)) (* x x))
= ((lambda (x) (* x x)) 3)

ML is a pure functional language which is strongly typed. It is impossible in ML to pass an argument of the wrong type to a function.

In ML, lists are strongly-typed. In particular the elements of a list must have the same type (compare with an array).

e.g. Scheme (1 2 3)
ML [1, 2, 3]

A list in Scheme like (1 a 6) has no counterpart in ML.

The semantics of ML are essentially the same as Scheme.

length :

```
fun length (L) =
    if L = [] then        <-- empty list
        ∅
    else
        1 + length (tl (L));
                ^
                same as cdr in Scheme
```

ML also has pattern parameters. We can rewrite the function using a set of cases.

```
fun length [] = ∅
  | length (x::xs) = 1 + length xs;
```

"or" → `|`

`(x::xs)` : pattern

`x::xs` matches `[1,2,3]` with `x = 1, xs = [2,3]`

matches `[[1,2],[2,3]]` with `x = [1,2], xs = [[2,3]]`

matches `[1]` with `x = 1, xs = []`

does not match `[]`

member :

```
fun member x L =
    if L = [] then
        false
    else if x = hd(L) then
            true
        else
            member x tL(L);
```

alternatively :

```
fun member x [] = false
  | member x (x::_) = true
  | member x (_::xs) = member x xs;
```

e.g. member 2 [1,2,3]
= member 2 [2,3]
= true

Two ways of writing functions:

$f(a, b)$

or $f\ a\ b$ ← currying of functions

compare with $\lambda x, y \ldots$ or $\lambda x \lambda y \ldots$