Polymorphic Types → Well-typed!

points as [ ]

datatype ilist = nil | :: of int * ilist

datatype 'a list = nil | :: of 'a * 'a list

pronounced alpha infix ::

α type variable

same: nil : 'a list, [ ] : 'a list

|:: [ ]
int 'a list

So here 'a is instantiated as int.

true :: [ ] So here 'a is bool.

[ ] :: [ ] 'a list list ⟷ [[ ]] So 'b is 'a list
same as
'a list 'b list

datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

val E = Empty : 'a tree

val T = Node (E, true, E) T: bool tree E: 'a tree

not bool tree!

```
let
    val E = Empty
    val T = Node (E, true, E)
in
    E
end
```

Error!

```
val Node (e, _, _) =
let
    val E = Empty
    val T = Node (E, true, E)
in
    T
end
```

↳ e : bool tree

```
(* trav : 'a tree -> 'a list *)
fun trav (Empty : 'a tree) : 'a list = [ ]
  | trav (Node (l, x, r)) = (trav l) @ (x :: (trav r))
```

```
(* zip : 'a list * 'b list -> ('a * 'b) list *)
fun zip ([ ] : 'a list, _ : 'b list) : ('a * 'b) list = [ ]
  | zip (_, [ ]) = [ ]
  | zip (x :: xs, y :: ys) = (x, y) :: zip (xs, ys)
```

```
datatype ('a, 'b) union = A of 'a | B of 'b
```

t Option

```
datatype 'a option = NONE | SOME of 'a
val L = zip ([1, 2, 3, 4], ["a", "b", "c", "d"])
    [(1, "a"), (2, "b"), (3, "c"), (4, "d")] / L
```

```
(* lookup : ('a * 'a -> bool) * 'a * (('a * 'b) list) -> 'b option *)
```

```
fun lookup (_ : 'a * 'a -> bool, _ : 'a, []: ('a * 'b) list) : 'b option = NONE
  | lookup (eq, x, (a,b)::rest) = if eq(x,a) then SOME(b) else lookup(eq, x, rest)
```

eg. lookup (op =, 3, L) → SOME("c") op = : int * int → bool
lookup (op =, 5, L) → NONE (op =)

Type Inference

Given an expression e, ML will try to determine the most general type (mgt) t for e, given all constraints in the code.

t is the mgt for e if e:t and whenever e:t' then t' is an instance of t.

instance means any type variables in t are the same as more specific in t'.

'a * 'a is an instance of 'a * 'b (but not the other way around)

([], []) : 'a list * 'b list

```
fun f(x, y, z) = 2*(x+z)      f : int * 'a * int → int
```

there's no value in SML that has every type.

```
fun square x = x * x          square : int → int
fun sqrf (f, x) = square(f(x))   sqrf : ('a → int) * 'a → int
fun f x = f(f(x))             f : 'a → 'a
fun g x = g(x)                g : 'a → 'b
fun twice f = f(f o)          twice : ('a → 'a) → 'a
```

) ?

Function Application is left associative. So f g x ≅ (f g) x

```
fun id x = x                  id : 'a → 'a
```

id twice square : int

'a → 'a ('a → 'a) → 'a 'a → 'a

'a

('a → 'a) → 'a

twice id square : ✗ type check

('a → 'a) → 'a 'a → 'a int → int

int