Standard ML

Where ML came from i

- 1973: Robin Milner at Edinburgh creates ML as the “meta language” for the LCF theorem prover
- The goal: write proof tactics that can’t forge theorems, so the type system guards soundness
- That need produced Hindley-Milner type inference and exceptions as a control tool

Standardization and descendants i

- 1983-1990: ML splinters into dialects, then is unified as Standard ML, with *The Definition of Standard ML* (1990, revised 1997)
- Milner wins the Turing Award in 1991
- Implementations: SML/NJ, MLton, Poly/ML
- Heirs: OCaml, F#, and large parts of Haskell, Rust and Scala
- Still used in theorem provers (HOL, Isabelle) and verified compilers (CakeML)

Why start with values and types i

- Everything in SML is an expression that produces a value, and every value has a type
- The compiler infers types, so you rarely write them out
- Use cases: quick calculations, and reading the inferred type to check your own understanding of the code

Values and types i

```
val x = 40 + 2;;
val greeting = "hello" ^ ", " ^ "world";;
val pair = (1, "one");;
```

Why functions i

- Functions are ordinary values: you can name them, pass them and return them
- `fn x => ...` creates a function and `fun f x = ...` is shorthand for naming one
- Use cases: small reusable transformations, and composing behavior with `o`

Functions i

```
fun square x = x * x;;
square 12;;
val inc = fn n => n + 1;;
(square o inc) 4;;
```

Why recursion and patterns i

- There are no loops: repetition is recursion
- A function can be defined by clauses that match the shape of its argument
- Use cases: anything inductive, such as factorials, tree walks and parsers

Recursion i

```
fun fact 0 = 1
  | fact n = n * fact (n - 1);;
fact 10;;
```

Why lists and higher-order functions i

- Immutable singly linked lists are the default collection
- `map`, `filter` and `foldl` replace most hand-written loops
- Use cases: data pipelines, aggregating records, and transforming input before processing it

Lists i

```
val xs = [1, 2, 3, 4, 5];;
map (fn x => x * x) xs;;
List.filter (fn x => x mod 2 = 0) xs;;
List.foldl op+ 0 xs;;
```

Why datatypes i

- A datatype lists every shape a value can take, and pattern matching handles each one
- The compiler warns when a case is missing
- Use cases: ASTs and interpreters, state machines, and protocol messages

Datatypes and pattern matching i

```
datatype shape = Circle of real | Rect of real * real;;
fun area (Circle r) = 3.14159 * r * r
  | area (Rect (w, h)) = w * h;;
area (Rect (3.0, 4.0));;
```

Why polymorphism i

- A function that doesn't inspect its argument gets a general type automatically
- `'a` is a type variable: one definition works for every type
- Use cases: generic containers, utilities like `swap` and `compose`, and library code

Polymorphism and inference i

```
fun swap (a, b) = (b, a);;
swap (1, "one");;
fun compose f g x = f (g x);;
```

Why options i

- There is no null: a value that may be absent has type `'a option`
- The type forces callers to handle `NONE`
- Use cases: lookups, searching, parsing, and anything partial

Options instead of null i

```
fun find p [] = NONE
  | find p (x :: xs) = if p x then SOME x else find p xs;;
find (fn x => x > 3) [1, 2, 3, 4, 5];;
find (fn x => x > 9) [1, 2, 3, 4, 5];;
```

Why exceptions i

- Options model expected absence, while exceptions model failures that cut across layers
- `raise` throws, and `handle` pattern matches on what was thrown
- Use cases: invalid input, resource errors, and early exit from deep recursion

Exceptions i

```
exception Empty;;
fun hd' [] = raise Empty
  | hd' (x :: _) = x;;
hd' [] handle Empty => ~1;;
```

Why modules i

- Structures group code, signatures describe an interface, and `:>` hides the representation
- It is the language's main tool for abstraction, in place of classes
- Use cases: abstract data types, swappable implementations, and large codebases with enforced boundaries

Modules i

```
signature STACK = sig
  type 'a t
  val empty : 'a t
  val push : 'a * 'a t -> 'a t
  val pop : 'a t -> ('a * 'a t) option
end;;
structure Stack :> STACK = struct
  type 'a t = 'a list
  val empty = []
  fun push (x, s) = x :: s
  fun pop [] = NONE
    | pop (x :: s) = SOME (x, s)
end;;
Stack.pop (Stack.push (1, Stack.empty));;
```

Try it yourself i

```
(* your code here *)
```