

Foundations of Computer Science

Layer	Description
Transistors	On the smallest scale, computers are turning transistors on and off.
Microcontroller	The Raspberry Pi Pico has millions of transistors .
Motherboard	Contains multiple CPUs all trying access shared resources (RAM).
Devices	The Apple Vision Pro contains lots of sensors and chips fit in a small box.
Supercomputer	Thousands of CPUs/GPUs working together, connected to internet and storage.
The user	Computers in data centres are rented out to users doign all sorts of stuff, e.g. doing AI research, watching Netflix, etc.

Abstraction

There's no way of understanding the whole tower at once: you cannot understand agentic AI in terms of transistors.

- With abstraction, you only understand the layer below.
- This is the “*What operations do I need to do the task?*” mentality of a programmer.

Definition

Abstraction barrier allows one layer to be changed without affecting levels above.

Representing Data

Definition

The concept of a **data type** involves

- How a value is represented inside the computer.
- The suite of operations (services) provided to the programmer.

How the data is represented may produce undesired results.

- The Y2K crisis.
- Floating point precision error.

Programming in OCaml

The goals of programming is to **describe a computation** so it can be done mechanically.

- Be efficient and correct.
- Allow easy modification: the effect of changes can be easily predicted.

Definitions

Keyword	What they do
Expressions	Compute values, <i>may</i> cause side effects.
Commands	Cause <i>only</i> side effects

Why OCaml?

- Interactive evaluation in **Jupyter notebooks** and in a **REPL**.
- Flexible and powerful notion of a data type.
- Hides underlying complexity - it throws an exception but never crashes, manages memory for us.
- Programs written can be understood/reasoned mathematically. (as there is no side effect)

Basics in OCaml

```

(* = Variable declaration *)
let pi = 3.1415926 (* val pi : float = 3.1415926 *)

(* = Function declaration *)
let area r = pi *. r *. r (* val area : float -> float *)

(* = Function invocation *)
area 2.0 (* - : float = 12.556 *)

(* = Recursive functions *)
let rec npower x n = (* npower : float -> int -> float = <fun> *)
  if n = 0 then 1.0
  else x *. power x (n - 1)

(* Type hints for the compiler *)
let side : float = 1.0 (* side : float *)
let square (x : float) = x *. x (* square : float -> float *)

(* If-Else *)
let rec power x n = (* power : float -> int -> float *)
  if n = 1 then x
  else if (n mod 2) == 0 then
    power (x *. x) (n / 2)
  else
    x *. power (x *. x) (n / 2)
(* this is a more efficient power function than npower *)

```

OCaml automatically infers types, but it does **not** implicitly convert types.

- Type inference by looking at the operations values, *. for float multiplication and * for integers.
- All branches of an if-else block must return value of the same type.

Type hints are useful to prevent OCaml from inferring all the wrong types when you make one small mistake.