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, *may* cause side effects.
Commands	Cause *only* 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 multplication 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.

Recursion and Complexity

Definition
Separating expressions from side effect is known as **functional programming**.

We can trace an expression, for example `power` function defined above.

```
⇒ power 2.0 12
⇒ power 4.0 6
⇒ power 16.0 3
⇒ 16 *. power 256.0 1
⇒ 16 *. 256.0
⇒ 4096.0
```

```
(* sums the first n integers *)
let rec nsum n =
  if n = 0 then 0
  else
    n + nsum (n - 1)
```

```
⇒ nsum 3
⇒ 3 + nsum 2
⇒ 3 + (2 + nsum 1)
⇒ 3 + (2 + (1 + nsum 0))
⇒ 3 + (2 + (1 + 0))
⇒ 6
```

Nothing can progress until the innermost sum is calculated. All the intermediate values have to be stashed onto the **program stack**. Evaluating `nsum 10000000` can cause a stack overflow.

Alternative Approach: Iterative Summing

```
let rec sum n total =
  if n = 0 then total
  else
    summing (n - 1) (n + total)
```

```
⇒ sum 3 0
⇒ sum 2 3
⇒ sum 1 5
⇒ sum 0 6
⇒ 6
```

The trace looks quite different.

- The total is known as an **accumulator**.
- Functions like this is called **tail recursive**.

Definition

In a **tail recursive** function, the recursive function call is the last thing the function does.

`nsum` is not tail recursive because it has to do the *add* operation after calling the function.

- `sum` won't stack overflow only if the compiler knows the function is tail recursive and optimises it.
- OCaml pops the function call off the stack before it finishes executing.

Downsides of Tail Recursion

- Extra variable needed, so easier to call the function incorrectly.
- Function is more complicated.

Don't use tail recursion with accumulator unless gain is significant.

Analysing Efficiency

```
let rec sillySum n =
  if n = 0 then 0
  else
    n + (sillySum (n - 1) + sillySum (n - 1)) / 2
```

`sillySum` is ran twice, as there may be side effects and the two functions may give different results.

- This is why pure functional evaluation is much simpler.
- Assign the value to a variable to avoid it being evaluated twice.

```
let rec sillySum n =
  if n = 0 then 0
  else
    let previousSum = sillySum (n - 1) in
      n + (previousSum + previousSum) / 2
```

Asymptotic complexity

Definition

Asymptotic complexity refers to how programs costs grow with increasing inputs.

E.g. space and time, the latter usually being larger than the former.

Definition

The **Big-O** notation is defined as $f(n) = O(g(n))$ provided that $|f(n) \leq c|g(n)|$ for large n.

Intuitively, consider the most significant term and ignore the constant coefficient or smaller factors.

Here are some interesting results:

- $O(\log n) = O(\ln n)$
- $O(\log n)$ is contained in everything, including $O(\sqrt{n})$
- An exponential algorithm can be faster than a linear algorithm for a particular input size interval.
- $O(n \log n)$ is called **quasi-linear**.

Simple Recurrence Relation

Set the time cost of base case $T(1) = 1$.

Recurrence relation	Time complexity
$T(n+1) = T(n) + 1$	$O(n)$
$T(n+1) = T(n) + n$	$O(n^2)$
$T(n) = T(n/2) + 1$	$O(\log n)$
$T(n) = T(n/2) + n$	$O(n \log n)$

Some examples in analysing time complexity.

```
let rec nsum =
  if n = 0 then 0
  else
    n + nsum (n - 1)
```

- $T(0) = 1$
- $T(n+1) = T(n) + 1$
- So $O(n)$

```
let rec nsumsum n =
  if n = 0 then 0
  else
    nsum (n - 1) + nsumsum(n -
1)
```

- $T(0) = 1$
- $T(n+1) = T(n) + n$
- So $O(n^2)$

```
let rec power x n =
  if n = 1 then x
  else if even n then
    power (x *. x) (n / 2)
  else
    x *. power (x *. x) (n / 2)
```

- $T(0) = 1$
- $T(n) = T(\frac{n}{2}) + 1$
- So $O(\log n)$

At each call n is halved, and add 1 as there is always some extra work (e.g. calling the function, if branch).