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))
```

$$\Rightarrow 6$$

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).

Lists

Definition

A **list** is a finite, ordered sequence of elements, all elements must have the same type.

List Primitives

There are only 2 kinds of lists, the 2 operations covers all possible lists.

```
[] (* nil : the empty list *)
x :: xs (* cons : put one element in front of the list *)
```

`[3; 5; 9]` is syntactical sugar for `3 :: (5 :: (9 :: []))`.

```
[3; 5; 9]
(* - : int list = [3; 5; 9] *)

[[3; 1]; [2]]
(* - : int list list = [[3; 1]; [2]] *)

(* concatenate two lists *)
[3; 5; 9] @ [2; 4]
(* - : [3; 5; 9; 2; 4] *)

(* the List library contains useful functions *)
List.rev [1; 2; 3]
(* - : [3; 2; 1] *)
```

Tuples

Definition
Tuples are fixed size and hetrogeneous sequences.

```
let pair = (1, true)
(* val pair : int * bool = (1, true) *)

(* you can do it without the brackets *)
let another_pair = 1, true, 3.2
(* val another_pair : int * bool * float = (1, true, 3.2) *)

(* take care not to use commas instead of semicolons *)
let list = [1, 2, 3]
(* val list : int * int * int list *)
```

Pattern Matching

All possible values that can be matched must be matched.

```
let null = function
  | [] -> true
  | _ :: _ -> false

let is_zero = function
  | 0 -> true
  | _ -> false
```

You can also pattern match parameters.

```
let hd = (x :: _) = x

hd [1] (* 1 *)
hd [] (* match error *)
```

In this case is better to use an option type.

Polymorphic Functions

The `List.tl` function returns the tail of a list

```
List.tl
(* - : 'a list -> 'a list = <fun> *)
```

An `'a` type (read: alpha type) means it can be of any type, but all elements of the list must be the same type.

More List Functions

```
let rec append = function
    | [], ys -> ys
    | x :: xs, ys -> x :: append xs ys
(* val append : a' list * a' list -> a' list *)
```

The match keyword keeps the reference to the original value.

```
let rec append xs ys =
  match xs, ys with
    | [], ys -> ys
    | x :: xs, ys -> x :: append xs ys
(* val append : a' list -> a' list -> a' list *)
```

Take and Drop

- `take` takes the first `i` items of a list.
- `drop` returns all the items that are not included in `take`

```
let rec take = function
  | [], _ => []
  | x :: xs, i =>
    if i > 0 then
      x :: take (xs, i - 1)
    else
      []
;;

let rec drop = function
  | [], _ -> []
  | x :: xs, i ->
    if i > 0 then
      drop (xs, i - 1)
    else
      x :: xs (* we could do this better using a match *)
;;
```

In the `drop` function:
- We advance the pointer as we go through the list.
- Then just returns the pointer where we stop.
- No memory is used.

In the `take` function has to construct a list from scratch.

Searching

Goal is to find x in a list $[x_1; \ldots; x_n]$

Name	Description	Cost
Linear search	Compare each element	$O(n)$
Oredred search	The list is bisected every time	$O(\log n)$
Indexed search	Create an index, e.g. a hash map	$O(1)$

Equality Test

The polymorphic equality operator = to compare integers, bools, floats but not functions.

Do not use ==

List Membership

```
let rec member x = function
  | [] -> false
  | y :: ys -> x = y || member x ys
```

The `||` is not a normal function, if the first case evaluates to true, it will not bother to evaluate the 2nd bit.

Zip and Unzip

```
let D in E
```

- Embeds declaration `D` within expression `E`
- Useful for performing intermediate computations within a function.

```
let rec zip = function
  | (x :: xs, y :: ys) ->
    (x, y) :: zip (xs, ys)
  | _ -> []
;;

let rec unzip pairs = function
  | [] -> []
  | (x, y) :: pairs ->
    let xs, ys = unzip pairs in
    (x :: xs, y :: ys)
;;
```

If we redo that in an iterative algorithm.

```
let rec unzipRev pairs = function
 | [], xs, ys -> xs, ys
 | (x, y) :: pairs, xs, ys ->
  unzipRev (pairs, x :: xs, y :: ys)
```

- In `unzip`, we traverse to the end then build up the list.
- In `unzipRev` we start building up the list right away.

That's why their order is different.

Sorting

Sorting is a key part of many other algorithms:

- **Searching** is much easier in a sorted list.
- **Merging** is also much easier.
- **Finding duplicates** is much easier as they would be next to each other in a sorted list.
- **Inverting tables** (??)
- **Graphics algorithms**: don't need to check for collision between every object. If the objects are sorted by distance, objects far away don't need to be checked.

Time Complexity of Sorting

Definition

In a **comparison sort**, the only way we can sort is by taking 2 times and comparing them: bigger/smaller than or equal.

We are limiting ourselves to comparison sort.

- There are $n!$ permutations of n elements.
- Each comparison eliminates half of the permutations $n^{C(n)} = n!$
- Therefore at best $C(n) \geq \log n! \sim n \log n + 1.44n$

Insertion Sort

```
let rec ins x = function
  | [] -> [x]
  | y :: ys ->
      if x <= y then
        x :: y :: ys
      else
        y :: ins x ys
;;

let rec insort = function
  | [] -> []
  | x :: xs ->
      ins x (insort xs)
;;
```

- The helper function inserts an item to a sorted list, on average the item is inserted to the middle of the list, so $O(n)$.
- `insort` has cost $O(n^2)$, much worse than $O(n \log n)$.

Tracing

We can trace a **monomorphic function** with

```
#trace insort
```

This prints out the function's arguments and return values.

Quicksort

1. Choose a pivot a, e.g. the head of list.
2. **Divide**: Partition the input into 2 sublists.
   - Those $\leq a$
   - Those $> a$
3. **Conquer**: recursively sort both sublists.
4. **Combine**: append the two lists together.

Ideally the two lists should be the same length, where we have $O(n \log n)$. If a sorted list is used, then the worst case $O(n^2)$. Since real data is often unsorted, we have average case $O(n \log n)$.

```
let rec quick = function
  | [] -> []
  | x :: xs ->
      let rec part l r = function
        | [] -> (quick l) @ (quick r)
```

```
        | y :: ys ->
            if y <= x then
              part (y :: l) r ys
            else
              part l (y :: r) ys
      in part [] [] xs
```

Merge Sort

```
let rec merge = function
  | [], ys -> ys
  | xs, [] -> xs
  | x :: xs, y :: ys ->
      if x < y then
        x :: merge xs (y :: ys)
      else
        y :: merge (x :: xs) ys
;;

let rec mergesort = function
  | [] -> []
  | xs ->
    let k = (List.length xs) / 2 in
    let l = take k xs in
    let r = drop k xs in
    merge (mergesort l) (mergesort r)
```

Merge sort has a worst case of $O(n \log n)$, but have a space complexity of $O(n \log n)$ due to the extra function calling we have to do.