

## 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 <b>transistors</b> .
Motherboard	Contains multiple <b>CPUs</b> all trying access shared resources (RAM).
Devices	The Apple Vision Pro contains lots of <b>sensors</b> and <b>chips</b> fit in a small box.
Supercomputer	Thousands of <b>CPUs/GPUs</b> working together, connected to internet and storage.
The user	Computers in <b>data centres</b> 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.

## 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

<code>let rec sum n total =</code>	⇒ sum 3 0
<code>  if n = 0 then total</code>	⇒ sum 2 3
<code>  else</code>	⇒ sum 1 5
<code>    summing (n - 1) (n + total)</code>	⇒ 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 heterogeneous 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  $[x_1; \dots; x_n]$

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.

---

## Datatypes and Trees

Using custom data types improves abstraction of data away from the details of representation.

- We can pack values into a tuple, but we can easily create **invalid values**.
- We want to create abstraction that is impossible to make mistakes like that.

```

let wheels = function
| "bike" -> 2
| "motorcar" -> 2
| "car" -> 4
| "lorry" -> 4

```

We want to make **illegal state** unrepresentable: a program with invalid states is rejected at compile time.

## Enumeration Types

```

type vehicle =
| Bike
| Motorbike
| Car
| Lorry

```

Instead of representing any string, it can only contain the 4 constants defined.

- The constants are the **constructors** of the vehicle type.
- This is more **memory efficient** than using strings.
- Adding new vehicle types is straightforward.

Even though the num is still represented as integers internally, we cannot compare it with any other data types. We have created a type that is *disjoint from any other types*.

## Generalisation

Data can be store alongside each variant.

```

type vehicle =
| Bike
| Motorbike of int
| Car of bool
| Lorry of int

```

```
let b = Motorbike 123
```

The Motorbike constructor has type `int -> vehicle`.

### Pattern Matching

```
let wheels = function
| Bike -> 2
| Motorbike _ -> 2
| Car true -> 3
| Car false -> 4
| Lorry w -> w
```

### Polymorphic Types

Types can have type parameters.

```
type 'a option =
| None
| Some of 'a

type ('a, 'b) result =
| Ok of 'a
| Error of 'b
```

### Exceptions

#### Definition

An exception is raised when something we weren't expecting happened.

- An exception is handled by doing an alternative computation.
- If not handled, the exception will go up to the top of the program and it will crash.

Note the alternative expression must return the same type as the original.

```
exception Fail
exception NoChange of int
```

```
raise Fail
raise NoChange 123
```

- `Fail` and `NoChange` are constructors for the `exc` type. They have type `exc` and `int -> exc` respectively.
- Since exceptions are values, we can pattern match on them.

```
try
(* do stuff *)
with
| NoChange -> (* alternative computation *)
```

`raise` dynamically jumps to the nearest `try/with` handler that matches the exception, it is can be used to jump to another part of the code.

### Recursive Datatypes

`list` is built into the language because the list syntax is helpful, but underneath it is just using another data type.

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Types are recursive by default, because we often want our types to be recursive. In fact, we need to use `nonrec` to disable this behaviour specifically.

## Binary Tree

If the list has 2 tails, we have a binary tree.

```
type 'a tree =
  | Lf
  | Br of 'a * 'a tree * 'a tree
```

Basic functions on trees includes:

```
let rec count = function
  | Lf -> 0
  | Br (_, l, r) -> 1 + count l + count r

let rec depth = function
  | Lf -> 0
  | Br (_, l, r) -> 1 + max (depth l) (depth r)
```

---

## Option or Exception?

Using the option type for fallible functions instead of using exceptions

- Pros: the detail that it can fail is included in the type signature of the function.
- Cons: if the exception is very rare, it is annoying to have a deeply nested match structure.

## Dictionaries

### Definition

**Dictionary** attaches values to identities (keys).

Operation	Description
lookup	Find an item in the dictionary.
update/insert	Replace/store an item in the dictionary.
delete	Remove an item from the dictionary.
empty	Creates the null dictionary with no keys.

## Implementation with Association List

`exception Missing`

```
let rec lookup a = function
  | [] -> raise Missing
  | (k, v) :: ps when x = a -> v
  | _ :: ps -> lookup a ps

let rec update l k v = (k, v) :: l
```

- lookup is  $O(n)$  because we have to go through each entry to find the key.
- update is  $O(1)$  as we are only shadowing the previous value.

**Note**

If the `remove` function only removes the first match in the list, it will uncover the previous value.

The old version of the association list still exist, we can look at it if we have a pointer to that list.

This is known as a **persistent data structure** as we are not modifying the original data structure.

**Implementation with Binary Search Tree**

- If the tree is balanced then lookup is  $O(\log n)$ .
- If unbalanced then lookup can be  $O(n)$ .

```
let rec lookup a = function
| Lf -> raise Missing
| Br ((k, v), l, r) when k = a -> v
| Br ((k, v), l, r) when k < a -> lookup a l
| Br ((k, v), l, r) -> lookup a r

let rec update a b = function
| Lf -> Br ((a, b), Lf, Lf)
| Br ((k, v), l, r) when k = a -> Br ((a, b), l, r)
| Br ((k, v), l, r) when k < a -> update a l
| Br ((k, v), l, r) -> update a r
```

Now removing a node does not uncover the previous value. If we have a pointer to the old tree, we can still see previous values in the dictionary.

**Tree Traversal****Definition**

The goal of **tree traversal** is to visit every node.

- Pre-order: visits the label first.
  - Gives a way to copy the elements of a tree.
  - Conversely post-order is a good way to deconstruct the tree.
- In-order: visits the label midway.
  - Using it on a binary search tree is known as **treesort**.
- Post-order: visits the label last.
  - Linearise an **expression tree** into RPN.

The other possibilities are breadth-first and depth-first traversal, but they are quite hard to implement as they are not recursive algorithms.

**Functional Arrays**

Arrays update in-place, they are **imperative** and **mutable** data structures.

We want to keep the core of the system mutation free, so it is easier to analyse. E.g.

- It is much easier for a multi-core system to work on the same piece of immutable data.
- But they are less efficient than e.g. quicksort which sorts in-place.

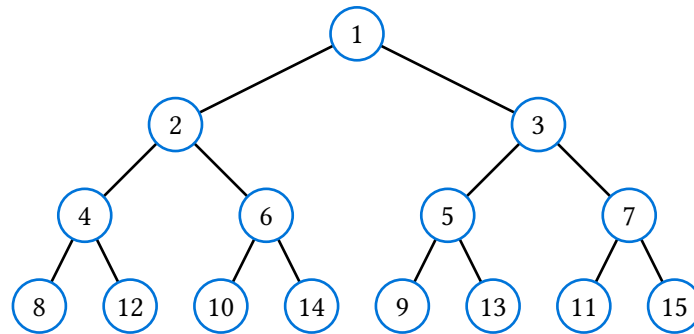
**Implementation**

Functional arrays store values in a **balanced tree**, so access time to each element is  $O(\log n)$ .

Say we want to access the node with index 12 (1100), we read the binary digits from right to left.

1. 0, so go to the left subtree.

2. 0, so go to the left subtree.
3. 1, so go to the right subtree.
4. 1 is the only digit remaining, the node we are currently at is the node we are searching for.



Note the numbers represents the index of the node, not its content.

```

exception Subscript (* index out of bounds *)
let rec lookup i = function
| Lf -> raise Subscript
| Br (v, l, r) when i = 1 -> v
| Br (v, l, r) when i mod 2 = 1 -> lookup (i / 2) r
| Br (v, l, r) -> lookup (i / 2) l

let rec update i v = function
| Lf when i = 1 -> Br (v, Lf, Lf)
| Lf -> raise Subscript
| Br (_, l, r) when i = 1 -> Br (v, l, r)
| Br (_, l, r) when i mod 2 = 1 -> update (i / 2) v r
| Br (_, l, r) -> update (i / 2) v l

```

---