Queens' College Cambridge

Foundations Of Computer Science

Alistair O'Brien

Department of Computer Science

April 7, 2020

Contents

1 OCaml

1.1 OCaml and Functional Programming

- OCaml is a functional programming language, tasks can be approached **mathematically**.
- OCaml provides **imperative** features such as
  - mutable arrays and variables (which are updated using the assignment command)
  - input and output commands

  Reason: pure functional methods such as *monads* reduce the clarity of the language.

Features

- Lists. Supports sequential access (scanning left from right).

  ```
  type 'a list = [] | (::) of 'a * 'a list
  ```

- Balanced tree. In theory, same efficiency as arrays, but in practice slower.
- First class objects: Objects that support all the operations generally available to other objects (such as being returned / applied to functions)
- Pattern matching: Pattern matching is the matching of objects against algebraic data type constructors.
- OCaml's type system is **polymorphic**

- Higher order functions: A function that either (or both) takes a function as an argument or returns a function. e.g. `List.map`, `List.fold`
- Exceptions can be raised and handled, $\implies$ OCaml cannot crash but enter an error state (which is contained).

Evaluation Of Expressions

- **Call-by-value** (strict evaluation):
  - Evaluation rule in OCaml.
  - Rule: To evaluate `f E`, first evaluate `E` then substitute the value into the body of `f`.
  - Conditionals have the rule: if `E_1` is true, evaluate `E_2` otherwise evaluate `E_3`.
- **Call-by-name**:
  - Rule: To evaluate `f E`, substitute `E` into the body of `f`. Then evaluate the body.
  - Good at reducing `f` if `f` doesn't dependent on `E`, but can lead to multiple evaluations of `E`.
- **Call-by-need** (lazy evaluation):
  - Similar to **call-by-name** but ensures `E` is evaluated at most once.
  - Substitutes pointers into the function body, if `E` is evaluated then the value is shared by the other pointers.
  - Pointer structure forms a directed graph.

1.1.1 Concrete Data Types

OCaml Type	Range
`int`	32 (or 64) bit two's complement integer
`float`	IEEE double-precision floating point
`bool`	A boolean, with literals `true` and `false`
`char`	An 8-bit character. Represented using `'...'`
`string`	A string. Represented using `"..."`
`unit`	Written as `()`. Used as a `null value`. `unit` is often returned by functions that produce side-effect e.g. `print_string : string -> unit`.

1.1.2 Operators

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>+</td><td>Integer addition</td></tr>
<tr><td>-(infix)</td><td>Integer subtraction</td></tr>
<tr><td>*</td><td>Integer multiplication</td></tr>
<tr><td>/</td><td>Integer division</td></tr>
<tr><td>mod</td><td>Integer modulo</td></tr>
<tr><td>-(prefix)</td><td>Negation</td></tr>
<tr><td>+.</td><td>Floating-point addition</td></tr>
<tr><td>-.</td><td>Floating-point negation</td></tr>
<tr><td>*.</td><td>Floating-point multiplication</td></tr>
<tr><td>/.</td><td>Floating-point division</td></tr>
<tr><td>**</td><td>Floating-point exponentiation</td></tr>
<tr><td>@</td><td>List concatenation</td></tr>
<tr><td>^</td><td>String concatenation</td></tr>
<tr><td>=</td><td>Equality</td></tr>
<tr><td><></td><td>Inequality</td></tr>
<tr><td><</td><td>Less than</td></tr>
<tr><td><=</td><td>Less than or equals</td></tr>
<tr><td>></td><td>Greater than</td></tr>
<tr><td>>=</td><td>Greater than or equals</td></tr>
<tr><td>&&</td><td>Boolean and</td></tr>
<tr><td>||</td><td>Boolean or</td></tr>
<tr><td>not(prefix)</td><td>Boolean not</td></tr>
</table>

1.1.3 Local declarations

- A **declaration** is the process of assigning a name to an expression.
- Local declarations in OCaml begin with the `let` keyword followed by the identifiers of the value. e.g.

```
let <pat1> = e1 and ... and <patn> = en [in u]
```

- Value identifiers are referred to as **variables** (bad use of the word...).
- A variable can be redeclared but not updated.
- The primitive types of values are `int, float, bool, char, string`.
- Identifiers can be
  - **alphabetic**: Starts with a letter, followed by zero or more letters, digits, underscores or aspostrophe.
  - **symbolic**: One or more of `!,%,&,$,#,+,-,*,/,:,<,=,>,?,@,\,~,‘,^`

1.1.4 Pattern Matching

- A **pattern** is defined as an expression consisting of variables, constructors and wildcards. Constructors are
  - literals (e.g. ints, floats, strings, etc)
  - algebraic data type value constructors
- **Pattern matching** is the processed of checking whether a value matches a given pattern. (they are usually matched using FSMs).
- e.g. Consider matching on a `maybe` value, then

```
let f x = match x with
    | Nothing -> raise (Invalid_argument "f")
    | Just x -> x
```

- Patterns have the general grammar

```
<pat> ::= _
        | <var>
        | <ValCons> <pat>
        | <ValCons> (<pat>, <pat>, ..., <pat>)
```

1.1.5 Control Structures

Sequence

- The sequence expression has the form

```
E_1 ; ... ; E_n
```

- The expression first evaluates `E_1`, then `E_2`, ..., then `E_n` and returns the value of `E_n`.
- The type of `(E_1 : t1) ; ... ; (E_n : tn)` is `tn`.

Conditional

- Conditional expressions have the form

```
if E_1 then E_2 else E_3
```

where `E_1` is an expression of type `bool`, and `E_2, E_3` are expressions of type `'a`. The type of conditional is `'a`.

- `else E_3` can be omitted, in which case it defaults to `else ()` (useful for imperative programming)

Match Expression

- The match expression has the form

```
match E_1, ..., E_n with
    | <pat11>, ..., <pat1n> [when cond_1] -> e_1
    | ...
    | <patm1>, ..., <patmn> [when cond_m] -> e_m
```

Exception Handling

- An **exception** is an datatype, that once `raise`d, disrupts the normal flow of the program's execution
- They are defined using the following syntax:

  ```
  exception E [of t]
  ```

  where `E` is an exception constructor name and `t` is some optional type. e.g.

  ```
  exception Problem of string
  ```

- To raise an exception value `e`, use the `raise` function.
  e.g. `raise $ Invalid_argument "foldr1"`
- To handle an exception, we use the following syntax

  ```
  try E_0 with
      | pat1 -> E_1
      .
      .
      .
      | patn -> E_n
  ```

  where `E_0` is some expression that might raise an exception. If no exception is raised, then the `try` expression evaluates to `E_0`. Otherwise, if it does raise some exception value `v`, the value `v` is matched against the patterns. e.g.

  ```
  try hd [] with
      | Invalid_argument s -> s
  ```

1.2 Functions

1.2.1 Function Definitions

- Non-recursive functions are defined as:

  ```
  let f x1 x2 ... xn = e
  ```

  Recursive functions require the `rec` keyword:

  ```
  let rec f x1 x2 ... xn = e
  ```

- Wee can use type annotations to explicitly denote types. e.g. `let f (x : int) : int = x + 2`.

- Multually recursive functions are defined with the `and` keyword e.g.

  ```
  let rec even n = n = 0 || odd (n - 1)
  and     odd  n = n = 1 || even (n - 1)
  ```

- Function types are

  ```
  t1 -> t2 -> ... -> tn -> u
  ```

  where `x1 : t1`, ..., `xn : tn` are the metavariables indicating types and `e : u`.

- The type operator `->` is right associative (curried functions).

1.2.2 Anonymous Functions

- An anonymous function is a function definition that is not bound to an identifier.

- In OCaml, there are two different types of syntax for anonymous functions:

  1. `function` creates an anonymous function that can match 1 variable to n patterns using

```
function
| <pat1> [when cond1] -> E_1
.
.
.
| <patn> [when condn] -> E_n
```

2. `fun` creates an anonymous function that can match n variables to 1 pattern each

```
fun <pat1> ... <patn> -> E
```

- They're often arguments being passes to higher-order functions, or used for constructing the results of a higher-order function.

1.2.3 First-Class Objects

- A first-class object is an entity which supports all the operations generally available to other objects. e.g. being passes as an argument, returned from a function, etc.
- OCaml functions are first-class objects.

1.2.4 Currying and Partial Application

- Currying is the process of transforming a function that takes multiple arguments in a tuple, into a function that takes a single argument and returns another function. e.g. `f : a -> (b -> c)` is the curried form of `g : (a,b) -> c`
- Note that `->` is right associative.
- Curried functions are more convenient because it allows partial application.
- Partial application is where less arguments than the full number of arguments are applied to a function. e.g.

```
let add x y = x + y
let add_one = add 1
```

Note that function application is left associative e.g.

```
f E_1 E_2 ... E_n = (... ((f E_1) E_2) ...) E_n
```

1.2.5 Higher Order Functions

- A **higher-order function** is a function that takes other functions as arguments or returns a function as a result. e.g.

```
(* curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c *)
let curry f a b = f (a, b)
(* uncurry : ('a -> 'b -> 'c) -> ('a * 'b -> 'c) *)
let uncurry f (a, b) = f a b
```

1.2.6 Polymorphic Functions

- OCaml's functions and abstract data types support parametric polymorphism (and subtype polymorphism).

- Parametric polymorphism refers to when the type of a value contains one or more type parameters. e.g.

```
let id x = x
- val id : 'a -> 'a = <fun>
```

1.2.7 Operators as Functions

- In OCaml, infix operators can be used as prefix functions by surrounding them with parentheses:

```
( + );;
- : int -> int -> int = <fun>
```

- We can define our own infix operators using the same syntax:

```
let ( <.> ) f g = fun x -> f (g x)
```

1.3 Imperative Features

1.3.1 References

- In OCaml we use a data abstraction of the computer's memory, via references to memory cells. A reference with initial contents of E is created using ref E. ref is a function of type 'a -> 'a ref.
- The function ! applied to the reference p is used to return the current contents of the reference p. This operation is known as dereferencing. It has the type 'a ref -> 'a.
- To update the contents of a reference, the assignment operator := is used. p := E assigns the contents of p to the value of E. := has the type 'a ref -> 'a -> ().
- Two references are said to be equal if and only if they have the same contents.

1.3.2 Control Structures

- There are two iterative control structures in OCaml:
  - The for loop, with the syntax

    ```
    for <name> = E_1 to E_2 do
        E_3
    done
    ```

    where E_1 is the initial value <name>. The body of the for loop E_3 is evaluated until <name> = E_2. The entire type of the for loop structure is (). e.g.

    ```
    for x = 1 to 4 do print_int x; print_newline () done
    ```

    prints the integers 1,2,3,4.

- The while loop, with syntax

```
while B do
    E
done
```

  This evaluates the Boolean expression `B`. `E` is evaluated zero or more times until `B` evaluates to `false`.

- For branching structures, we `if - then - else` and the `match` constructs.

2 Concepts in OCaml

2.1 Data Structures

2.1.1 Algebraic Data Types

- An algebraic data type in OCaml has the following form:

```
type ('a, 'b, ..., 'c) alg_data_type =
      Constructor1
    | Constructor2
      .
      .
      .
    | ConstructorK
    | ConstructorK' of t11 * t12 * ... * t1m
      .
      .
      .
    | ConstructorN of tn1 * tn2 * ... * tnm
```

  where `alg_data_type` is the name of the type of arity > 0, `'a, 'b, ..., 'c` are distinct type parameters and `Constructor1, ...` are the value constructors that describe the ways in which the values of the `alg_data_type` type can be constructed.

- Algebraic data types can be polymorphic, via the use of type parameters.

  e.g. Consider the `maybe` type (often called `option`).

```
type 'a maybe = Nothing | Just of 'a
```

We can then have `int maybe, float maybe, string maybe, ...` types.

- All non-recursive algebraic data types can be represented using the `disjoint_sum` data type.

```
type ('a,'b) disjoint_sum = Ln1 of 'a | Ln2 of 'b
```

  Consider the type `maybe`, then

```
let nothing = Ln1 ()
let just x = Ln2 x
```

  Provided we use a unique `disjoin_sum` value in each of our value constructor representation functions then we can represent the original non-recursive type.

- Algebraic data types can be recursive. e.g. `list` type.

```
type 'a list = [] | (::) of 'a * 'a list
```

2.1.2 Lists

- In OCaml, "lists" are singly-linked lists consisting of a finite sequence of elements.
- They are defined as

```
type 'a list = [] | (::) of 'a * 'a list
```

- There are three syntactic forms for building lists:

```
[] (*nil*)
e1 :: (* cons *) e2
[e1; e2; ...; en] <=> e1 :: e2 :: ... :: en :: []
```

- The cons operation is right associative.
- All elements in a list must have the same type. (However, the use of ADTs may subvert this condition).

```
module type List = sig
    exception Empty
    type 'a list

    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list

    val is_empty : 'a list -> bool
    val empty : 'a list

    val foldl : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
    val foldr : ('a -> b' -> 'b) -> 'b -> 'a list -> 'b

    val zip : 'a list -> 'b list -> ('a * 'b) list
    val unzip : ('a * 'b) list -> 'a list * 'b list

    val length : 'a list -> int
    val rev : 'a list -> 'a list
    val (@) : 'a list -> 'a list -> 'a list

    val map : ('a -> 'b) -> 'a list -> 'b list
    val filter : ('a -> bool) -> 'a list -> 'a list

end

module List : List = struct
    type 'a list = [] | (::) of 'a * 'a list
    exception Empty

    let hd xs = match xs with
        | x :: _ -> x
        | _ -> raise Empty
```

```
    let tl xs = match xs with
        | _ :: xs -> xs
        | _ -> raise Empty

    let empty = []

    let is_empty = match xs with
        | [] -> true
        | _ -> false

    let rec foldl f acc xs = match xs with
        | [] -> acc
        | x :: xs -> foldl f (f acc x) xs

    let rec foldr f acc xs = match xs with
        | [] -> acc
        | x :: xs -> f x (foldr f acc xs)

    let rec zip xs ys = match xs, ys with
        | x :: xs, y :: ys -> (x, y) :: zip xs ys
        | _, _ -> []

    let unzip = foldr (fun (x, y) (xs, ys) -> (x :: xs, y :: ys)) ([], [])

    let length = foldl (fun acc _ -> acc + 1) 0

    let cons x xs = x :: xs
    let rev = fold cons []
    let (@) xs ys = foldr cons ys xs

    let map f = foldr (fun x xs -> f x :: xs) []
    let filter f = foldr (fun x xs -> if f x then x :: xs else xs) []

end

(* useful list functions *)

let rec nth xs n = match xs, n with
```

```
    | [], _ -> raise Empty
    | _, n when n < 0 -> raise (Invalid_argument "nth")
    | x :: _, 0 -> x
    | x :: xs, n -> nth xs (n - 1)

let flatten = foldr (@) []
let exists f = foldl (fun acc x -> f x || acc) false
let all f = foldl (fun acc x -> f x && acc) true
let member x = exists ((=) x)
let partition f = foldr (fun x (xs, ys) -> if f x then ) ([], [])
```

2.1.3 Tuples

- Pairs are tuples of 2 values e.g. `(e1, e2) : 'a * 'b`
- n-tuples are created by an expression of the form `(e1, e2, ..., en)`.
  In ML, tuples have the form `(e1, ..., en) = (e1, (e2, (..., (en-1, en) ... )))`.
  Hence n-tuples can be implemented as extended pairs.
- A 0-tuple (denoted `()`) is a `null` value of type `unit`. Often used in function that produce side-effects, such as `print_string : string -> unit`.

2.1.4 Binary Trees

- A binary tree is a rooted tree (a connected simple graph with no cycles) where each vertex has at most two *children* (the left and right child).
- The number of nodes n on a binary tree of height $h \geq 0$ is

$$h + 1 \leq n \leq 2^{h+1} - 1.$$

```
module Tree = struct
    type 'a tree = EmptyTree | Vertex of 'a * 'a tree * 'a tree
    exception Empty

    let rec max t = match t with
        | EmptyTree -> raise Empty
        | Vertex (v, _, EmptyTree) -> v
```

```
          | Vertex (_, _, r) -> max r

      let rec min t = match t with
          | EmptyTree -> raise Empty
          | Vertex (v, EmptyTree, _) -> v
          | Vertex (_, l, _) -> min l

      let rec insert t v = match t with
          | EmptyTree -> Vertex (v, EmptyTree, EmptyTree)
          | Vertex (u, l, r) when v = u -> Vertex (v, l, r)
          | Vertex (u, l, r) when v < u -> Vertex (u, insert l v, r)
          | Vertex (u, l, r) -> Vertex (u, l, insert r v)

      let rec delete t v = match t with
          | EmptyTree -> EmptyTree
          | Vertex (u, l, r) when v < u -> Vertex (u, delete l  v, r)
          | Vertex (u, l, r) when v > u -> Vertex (u, l, delete r v)
          | Vertex (u, l, r) ->  match t with
              | Vertex (_, EmptyTree, EmptyTree) -> EmptyTree
              | Vertex (_, EmptyTree, r) -> r
              | Vertex (_, l, EmptyTree) -> l
              | Vertex (_, l, r) ->
                  let v' = min r in
                  let r' = delete r v'
                  in Vertex (v', l, r')

      let rec search t v = match t with
          | EmptyTree -> raise Empty
          | Vertex (u, _, _) when v = u -> t
          | Vertex (u, l, _) when v < u -> search l v
          | Vertex (u, _, r) -> search r v


  end
```

- A *dictionary* is a collection of key-value pairs, such that each key appears at most once in the collection. An ordered dictionary is a dictionary in which keys have some total ordering.

```
module type Dictionary = sig
    type key
    type 'a dict
    exception NotFound

    val empty : 'a dict

    val insert : 'a dict -> key -> 'a -> 'a dict
    val search : 'a dict -> key -> 'a
end
```

- A *binary search tree* is a binary tree in which each vertex stores a key-value pair (k, v), such that for all keys k_l in the left subtree l satisfies $k_l < k$ (and vice-versa).
- A binary search tree implements an ordered dictionary.

```
module BST : Dictionary = struct
    include Tree

    type key = string
    type 'a dict = (key * 'a) tree
    exception NotFound

    let empty = EmptyTree

    let rec insert d k v = match d with
        | EmptyTree -> Vertex ((k, v), EmptyTree, EmptyTree)
        | Vertex ((k', v'), l, r) when k = k' ->
            Vertex ((k', v'), l, r)
        | Vertex ((k', v'), l, r) when k < k' ->
            Vertex ((k', v'), insert l k v, r)
        | Vertex ((k', v'), l, r) ->
            Vertex ((k', v'), l, insert r k v)
```

```
    let rec search t k = match t with
        | EmptyTree -> raise NotFound
        | Vertex ((k', v'), _, _) when k = k' -> v'
        | Vertex ((k', _), l, _) when k < k' -> search l k
        | Vertex ((k', _), _, r) -> search r k

end
```

2.1.5 Queues

- A queue is a first in first out (FIFO) data structure, where elements are removed from the head and inserted at the tail.
- Functional queue $x_1, \ldots, x_m, y_n, \ldots, y_1$ represented by a pair of lists

$$(\underbrace{[x_1, \ldots, x_m]}_{\text{front}}, \underbrace{[y_1, \ldots, y_n]}_{\text{rear}}).$$

- Enqueue to the rear of the queue and dequeue from the front.
- Amortized time per operation is $O(1)$.
- **Implementation**:

```
module type Queue = sig
    exception Empty

    type 'a queue
    val empty : 'a queue

    val is_empty : 'a queue -> bool

    val enqueue : 'a queue -> 'a -> 'a queue
    val dequeue : 'a queue -> 'a queue

    val hd : 'a queue -> 'a
end
```

```
module FunctionalQueue : Queue = struct
    exception Empty

    type 'a queue = 'a list * 'a list
    let empty = ([], [])

    let is_empty q = match q with
        | ([], []) -> true
        | _ -> false

    let norm q = match q with
        | ([], ys) -> (List.rev ys, [])
        | q -> q

    let enqueue (xs, ys) y = norm (xs, y :: ys)
    let dequeue q = match q with
        | (x :: xs, ys) -> norm (xs, ys)
        | _ -> raise Empty

    let hd q = match q with
        | (x :: _, _) -> x
        | _ -> raise Empty
end
```

2.2 Algorithms

2.2.1 Sorting

- **Problem Of Sorting**: Given sequence $\langle x_1, x_2, \ldots, x_n \rangle$. Return permutation of sequence $\langle x'_1, x'_2, \ldots, x'_n \rangle$ such that $x'_1 \leq x'_2 \leq \cdots \leq x'_n$ (monotonically increasing)
- **Insertion Sort**:
  - Tail recursive (or iterative) comparison sorting algorithm.
  - At each recursive call, we remove an element (the head) and insert it in the correct position in the sorted sublist.

```
let rec insert e xs = match xs with
    | [] -> [e]
    | x :: xs when e <= x -> e :: x :: xs
    | x :: xs -> x :: (insert e xs)

let rec insertion_sort xs = match xs with
    | [] -> []
    | x :: xs -> insert x (insertion_sort xs)
```

- **Analysis**:
  - **Worst Case**: Occurs when the list `xs` is reverse sorted. `insert` takes $\Theta(j)$ time for list `xs` of size j (in worst case). Hence
  $$T(n) = \sum_{j=1}^{n} \Theta(j) = \Theta(n^2).$$
  - **Average Case**: Average case time complexity is $\Theta(n^2)$. See algorithms notes for analysis.
- Variants such as *binary insertion sort* exist, reducing # of comparisons, but still have time complexity of $O(n^2)$

• **Quicksort**:
- Divide and conquer comparison sorting algorithm:
  - **Divide**: Partition the list to be sorted into two sublists around a pivot x, such that elements in the lower sublist satisfy $\leq x$ and elements in the upper sublist satisfy $> x$.
  - **Conquer**: Recursively sort the lower and upper sublists using quicksort
  - **Combine**: Combine the sorted sublists by appending them together.

```
let rec quicksort xs = match xs with
    | [] -> []
    | x :: xs ->
        let (l, r) = partition (sw (<=) $ x) xs in
        quicksort l @ x :: r
```

- **Analysis**:
  - **Worst Case**: Partition produces one sublist of length $n-1$ and one of length 0. `partition` takes $\Theta(n)$ time. So we have

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

    Hence $T(n) \in \Theta(n^2)$. See algorithms notes for formal analysis.
  - **Average Case**: The average case time complexity is $O(n \log_2 n)$. See algorithms notes for formal analysis.
- A tail recursive variant

```
let rec quicksort xs sorted = match xs with
    | [] -> sorted
    | x :: xs ->
        let (l, r) = partition (sw (<=) $ x) xs in
        quicksort l (x :: quicksort r sorted)
```

- **Merge Sort**:
  - Divide and conquer comparison sorting algorithm:
    - **Divide**: Divide the n element list `xs` into two sublists `l, r` of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
    - **Conquer**: Recursively sort the two sublists using merge sort.
    - **Combine**: Merge the two sorted sublists using `merge`.

```
let rec merge xs ys = match xs, ys with
    | [], 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 merge_sort xs = match xs with
    | [] -> []
    | [x] -> [x]
    | xs ->
```

```
let k = length xs / 2
and l = merge_sort (take xs k)
and r = merge_sort (drop xs k) in
merge l r
```

- **Analysis**:
  * Let $T(n)$ be the time cost function of `merge_sort` where n is the length of `xs`. We have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \underbrace{\Theta(n)}_{\text{merge}} & \text{otherwise} \end{cases} \cdot$$

    since `merge` performs $m' + n' - 1$ comparisons (in the worst case) where m' and n' are the lengths of `xs` and `ys` respectively. Hence $\Theta(n)$ time.
    So by master theorem, we have

$$T(n) \in \Theta(n \log_2 n).$$

- Merge sort's worst case doesn't depend on the initial permutation of `xs`, whereas quicksort does.

2.2.2 Traversals

- **Tree Traversal**: A form of graph traversal in which each vertex of the tree data structure is visited exactly *once*. Classified by order in which vertices are visited.

- **Depth-first Searches**. A depth-first search (DFS) starts at a root vertex and explores as far as possible along each branch of the tree before backtracking. General recursive pattern (for a binary tree) with root x:
  - (L) Recursively traverse x's left subtree l.
  - (V) Process the current vertex x.
  - (R) Recursively traverse x's right subtree r.

Common DFS traversals:

- **Pre-order** (VLR):
  * Process current vertex, recursively traverse l then recursively traverse r.
  * Produces a topologically sorted list

    ```
    let rec preorder x = match x with
        | EmptyTree -> []
        | Vertex (v, l, r) -> [v] @ preorder l @ preorder r
    ```

  * Tail-recursive variant removes append

    ```
    let rec preorder x acc = match x with
        | EmptyTree -> acc
        | Vertex (v, l, r) -> v :: preorder l (preorder r acc)
    ```

- **In-order** (LVR):
  * Recursively traverse l, process current vertex x then recursively traverse r.
  * In BST, in-order traversal produces a sorted list of vertex keys.

    ```
    let rec inorder x = match x with
        | EmptyTree -> []
        | Vertex (v, l, r) -> inorder l @ [v] @ inorder r
    ```

  * Tail-recursive variant

    ```
    let rec inorder x acc = match x with
        | EmptyTree -> acc
        | Vertex (v, l, r) -> inorder l (v :: inorder r acc)
    ```

- **Post-order** (LRV):
  * Recursively traverse l, recursively traverse r then process current vertex x.

    ```
    let rec postorder x = match x with
        | EmptyTree -> []
        | Vertex (v, l, r) -> postorder l @ postorder r @ [v]
    ```

    * Tail-recursive variant

```
let rec postorder x acc = match x with
    | EmptyTree -> []
    | Vertex (v, l, r) -> postorder l (postorder r (v :: acc))
```

  - **Analysis**: Worst case time complexity of implementations using `@` are $O(|V|^2)$ (due to append). Tail-recursive variants have a worst-case complexity of $\Theta(|V|)$.

- **Breadth-first Searches**. A Breadth-first search (BFS) starts at a root vertex and explores all of the neighbours at the current depth prior to moving onto the next depth level.

  Implementations :

  - **Naïve Implementation**. Use a list `xs` to store all vertices that will be visited. Each iteration removes a vertex from the head of the list and appends it's subtrees at the end of the list (FIFO ordering).

```
let rec bfs xs = match xs with
    | [] -> []
    | EmptyTree :: xs -> bfs xs
    | Vertex (v, l, r) :: xs -> v :: bfs (xs @ [l; r])

bfs : 'a tree list -> 'a list
```

  - **Queue Implementation**. We use a functional queue implementation with enqueue and dequeue operations with amortized costs $O(1)$.

```
open FunctionalQueue

let rec bfs q =
    if is_empty q then []
    else
        match hd q with
            | EmptyTree -> bfs (dequeue q)
            | Vertex (v, l, r) ->
```

```
                        v :: bfs (enqueue (enqueue (dequeue q) l) r)

bfs : 'a tree queue -> 'a list
```

Analysis: BFS to depth d with branching factor b (average degree) examines $O(b^d)$ vertices

$$n = 1 + b + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1} = \frac{b}{b-1}(b^2 - b^{-1}) \in O(b^d),$$

with time factor $\frac{b}{b-1}$.

- **Iterative deepening DFS**. Iterative deepening DFS (IDDFS) is a search algorithm in which DFS depth-limited algorithm is repeatedly run until the search goal (solution) is found. ‘’

  Analysis: The vertices at depth d are explored once, the vertices at $d-1$ are explored twice, . . . , the root vertex is explored $d+1$ times. So we have the following arithmetic-geometric sequence

$$\begin{aligned} b^d + 2b^{d-1} + 3b^{d-2} + \cdots db + (d+1) &= \sum_{k=0}^{d} (d+1-k)d^k \\ &= b^d \sum_{k=0}^{d} (d+1-k)d^{k-d} \\ &\leq b^d \sum_{k=1}^{\infty} k(b^{-1})^{k-1} \\ &= b^d \frac{\mathrm{d}}{\mathrm{d}(b^{-1})} \sum_{k=0}^{\infty} (b^{-1})^k \\ &= b^d \left(1 - \frac{1}{b}\right)^{-2} \in O(b^d) \end{aligned}$$

  For the space complexity, the `dfs` function stores a stack of maximum depth d, hence space complexity of IDDFS is $O(d)$.

  Advantages:

  – BFS complete search on infinite trees while having DFS space complexity.

  – Despite revisiting vertices, IDDFS is extremely efficient.

2.3 Sequences

- Implement lazy lists (sequences) in OCaml using delayed evaluation on the tail of the list.
- Delayed evaluation is implemented using a function `fun () -> e : () -> 'a` , delayed under a closure (not a lazy block). `e` is not evaluated until the function is called, thus delaying the evaluation of `e`.

```
module type Seq = sig
    type 'a seq
    exception Empty

    val hd : 'a seq -> 'a
    val tl : 'a seq -> 'a seq

    val empty : 'a seq
    val is_empty : 'a seq -> bool

    val map : ('a -> 'b) -> 'a seq -> 'b seq
    val filter : ('a -> bool) -> 'a seq -> 'a seq

    val (@) : 'a seq -> 'a seq -> 'a seq
    val interleave : 'a seq -> 'a seq -> 'a seq

end

module Seq : Seq = struct
    type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
    exception Empty

    let hd xs = match xs with
        | Cons (x, _) -> x
        | _ -> raise Empty

    let tl xs = match xs with
        | Cons (_, xf) -> xf ()
        | _ -> raise Empty
```

```
    let empty = Nil
    let is_empty xs = match xs with
        | Nil -> true
        | _ -> false

    let rec map f xs = match xs with
        | Nil -> Nil
        | Cons (x, xf) -> Cons (f x, fun () -> map f (xf ()))

    let rec filter f xs = match xs with
        | Nil -> Nil
        | Cons (x, xf) when f x -> Cons (x, fun () -> filter f (xf ()))
        | Cons (x, xf) -> filter f (xf ())

    let rec (@) xs ys = match xs with
        | Nil -> ys
        | Cons (x, xf) -> Cons (x, fun () -> (xf ()) @ ys)

    let rec interleave xs ys = match xs with
        | Nil -> ys
        | Cons (x, xf) -> Cons (x, fun () -> interleave ys (xf ()))

end
```