# Chapter 3: Data and Types

## 3.1 Lists

```
utop # [];;
- : 'a list = []
─( 09:21:57 )─< command 4 >──────────────────────
utop # [1,2,3];;
- : (int * int * int) list = [(1, 2, 3)]
─( 18:53:23 )─< command 5 >──────────────────────
utop # ["Hello", "world"];;
- : (string * string) list = [("Hello", "world")]
```

create lists, empty, int, string lists

```
utop # 1:: 2:: 3:: [];;
- : int list = [1; 2; 3]
```

another syntax to create lists

### 3.1.1 Building Lists

syntax:

- [ ] indicates an empty list, pronounced “nil”, empty list has type t list for any type of t
- e1 :: e2 indicates elements e1 to list e2, “: :” is pronounced “cons”, e2 means the rest of the list
- [e1; e2] is sugar for e1 :: e2 :: [];;

### 3.1.2 Accessing Lists

Pattern matching: allows users to break apart the list, powerful feature to accomplish many things

```
utop # match not true with
  | true -> "its true"
  | false -> "its false";;
- : string = "its false"
```

example: match not true with. Because the sentence “not true” doesn’t equal to the true condition, the false will execute

```
utop # match true with
 | true -> "its true"
 | falase -> "its false";;
- : string = "its true"
```
example: true or false

```
utop # let y =
match 42 with
| foo -> foo;;
val y : int = 42
```
example: match number with value

```
utop # let z =
match "hello" with
| "msg" -> 0
| _ -> 5;;
val z : int = 5
```
example: match else conditions with the match key. Matching “hello” to certain conditions, receives 5 for its “default” value, as “hello” doesn’t match with “msg”, so the last choice(_) is chosen. _ = all other options, except for option above

```
utop # let list =
match [1;2;3] with
| [] -> "its an empty list"
| _ -> "its a fulfilled list";;
val list : string = "its a fulfilled list"
```
example: match int list with two conditions, If its an empty list([]), will return “its an empty list”. If it’s not an empty list, return “it’s a fulfilled list”.

```
utop # let b =
match ["hello"; "world"] with
| [] -> "list is empty"
| h :: t -> h;;
val b : string = "hello"
```
example: match string list with condition . Because the string list isn’t empty, h(head), t(tail), h through t, return the h, which h is “hello”

```
utop # let rec sum lst =
match lst with
| [] -> 0
| h :: t -> h + sum t;;
val sum : int list -> int = <fun>
-( 20:02:55 )-< command 14 >
utop # sum [];;
- : int = 0
-( 20:04:32 )-< command 15 >
utop # sum [1;2;3];;
- : int = 6
```
recursion sum function, takes in list parameter, if its empty array then return 0. Otherwise, loop through h(head)

and t(tail) with recursion sum method. t(tail) = rest of the list. Can use #trace methodName;; to trace the recursion method's inputs and values.

```
utop # let rec length lst =
match lst with
| [] -> 0
| h :: t -> 1 + length t;;
val length : 'a list -> int = <fun>
-( 20:04:47 )-< command 17 >-----------
utop # length [];;
- : int = 0
-( 20:11:21 )-< command 18 >-----------
utop # length [1;2;3];;
- : int = 3
```

recursion length function, takes in list parameter, if its empty array then return 0. Otherwise, loop through h(head) and t(tail) with recursion length method, counting the number of index of t list.

```
utop # let rec append list1 list2 =
match list1 with
|[] -> list2
| h :: t -> h :: append t list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
-( 20:11:37 )-< command 20 >-----------
utop # append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
-( 20:19:57 )-< command 21 >-----------
utop # append [] [2;4;6];;
- : int list = [2; 4; 6]
```

recursion append function, combines two list together. If list1 is empty, simply return list2. Otherwise, loop through h(head) with recursion t(tail) for new loop

### 3.1.3 (Not) Mutating Lists

```
let inc_first lst =
  match lst with
  | [] -> []
  | h :: t -> h + 1 :: t
```

function to return the same list as its input list, but with the first element incremented by 1

### 3.1.4 Pattern Matching with Lists

**Syntax.**

```
match e with
| p1 -> e1
| p2 -> e2
| ...
| pn -> en
```

each of the clause pi -> e1 is called a branch or case of the pattern match

```
utop # List.hd [1;2;3;4];;
- : int = 1
─( 20:28:42 )─< command 1 >
utop # List.tl [3;5;6;7];;
- : int list = [5; 6; 7]
```

List.hd returns the head of the list. List.tl returns all the tails of the list.

### 3.1.5 Deep Pattern Matching

- `_ :: []` matches all lists with exactly one element
- `_ :: _` matches all lists with at least one element
- `_ :: _ :: []` matches all lists with exactly two elements
- `_ :: _ :: _ :: _` matches all lists with at least three elements

[]=

exactly _=at lease

### 3.1.6 Immediate Matches

- when a function immediately pattern-matches against the final argument/ branch, instead of writing extra code like “match parameter with”, simply write function

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

- >

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
```

## 3.2 Variants

Variant: a data type representing a value that is one of the many possibilities. Like Java Enums.
Constructors: the individual names of the value of a variant in OCaml.
Constructor name starts with an uppercase letter. Ex: Sun, Mon
Syntax:

type t = C1 | C2 | … | Cn ;;

Example:

```
utop # type week_days = Mon | Tue | Wed | Thur | Fri | Sat | Sun;;
type week_days = Mon | Tue | Wed | Thur | Fri | Sat | Sun
```

Pattern Matching:

```
let int_of_day d =
  match d with
  | Sun -> 1
  | Mon -> 2
  | Tue -> 3
  | Wed -> 4
  | Thu -> 5
  | Fri -> 6
  | Sat -> 7
```

### 3.2.1 Scope

- two types defined with overlapping constructor names, for example:

```
type t1 = C | D
type t2 = D | E
let x = D
```

The type definition defined later wins.

## 3.4 Records and Tuples

### 3.4.1 Records

```
type student = {
  name: string;
  year: int;
}
student
let tw = {
  name =  "Taylor Swift";
  year = 2000
}
```

```
utop # #use "example.ml";;
type student = { name : string; year : int; }
val tw : student = {name = "Taylor Swift"; year = 2000}
─( 06:18:06 )─< command 1 >──────────────────────────
utop # tw.name;;
- : string = "Taylor Swift"
```

similar to Java's objects and classes

```
type student ={
  name: string;
  year: int;
}
student
let rbg = {
  name= "Ruth Badger";
  year= 2000;
}
```

```
utop # {rbg with name="Ruth Badger 2"};;
- : student = {name = "Ruth Badger 2"; year = 2000}
─( 06:44:19 )─< command 2 >──────────────────────
utop # rbg;;
- : student = {name = "Ruth Badger"; year = 2000}
```

record copy. Syntax: {e for f1 = e1; f2=e2…} e(record variable), f1(fieldName), e1(new value for f1). NOTICE: doesn't change original record, example for syntax sugar. Good for record's one change of field. Constructs new record with new values. Record copying: simply coping all fields from one record, with a change of field value.

## 3.4.2 Tuples

```
type time = int * int * string
int * int * string
let t = (5, 20, "hello")
time
let s : time = (2, 7, "world")
```

```
utop # #use "example.ml" ;;
type time = int * int * string
val t : int * int * string = (5, 20, "hello")
val s : time = (2, 7, "world")
-( 06:25:10 )-< command 1 >
utop # t;;
- : int * int * string = (5, 20, "hello")
```

define tuples, aggregating data with unnamed components.

```
type product = string * int
string * int
let a = ("tooth paste", 5)
```

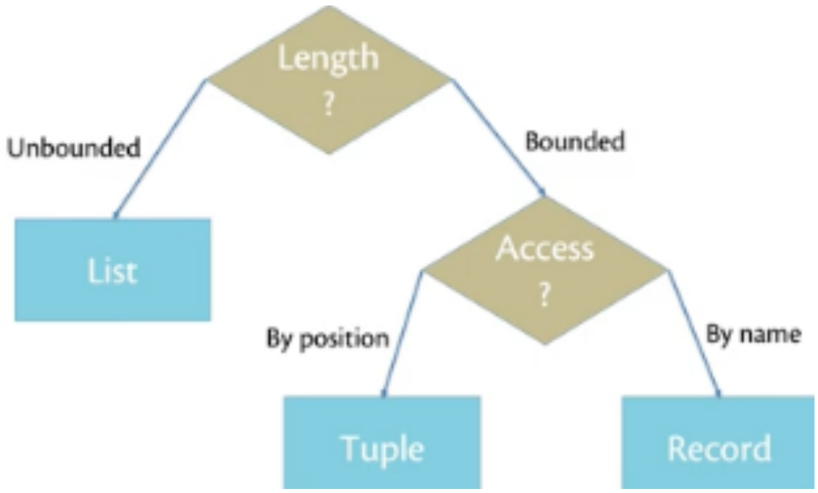
```
utop # fst a;;
- : string = "tooth paste"
-( 06:30:28 )-< command 2 >
utop # snd a;;
- : int = 5
```

defining tuples with pair values. Fst returns first component, snd returns second component. Built in function that only works on pairs. Tuple with two components = pairs. Tuple with three components = triple

```
utop # match (1, 2, 3) with (x, y, z) -> x + y + z;;
- : int = 6
```

pattern matching with building of types

### 3.4.3 Variants vs Tuples and Records



comparison between the datatypes

## 3.5 Advanced Pattern Matching

- `p1 | ... | pn`: an "or" pattern; matching against it succeeds if a match succeeds against any of the individual patterns `pi`, which are tried in order from left to right. All the patterns must bind the same variables.
- `(p : t)`: a pattern with an explicit type annotation.
- `c`: here, `c` means any constant, such as integer literals, string literals, and booleans.
- `'ch1'..'ch2'`: here, `ch` means a character literal. For example, `'A'..'Z'` matches any uppercase letter.
- `p when e`: matches `p` but only if `e` evaluates to `true`.

### 3.5.1 Pattern Matching with Let

Dynamic Semantics: define the run-time behavior of the program as its executed/evaluated

Static semantics: define the compile-time checking that is done to ensure the program is legal

**Dynamic semantics.**

To evaluate `let p = e1 in e2`:

1. Evaluate `e1` to a value `v1`.
2. Match `v1` against pattern `p`. If it doesn't match, raise the exception `Match_failure`. Otherwise, if it does match, it produces a set $b$ of bindings.
3. Substitute those bindings $b$ in `e2`, yielding a new expression `e2'`.
4. Evaluate `e2'` to a value `v2`.
5. The result of evaluating the let expression is `v2`.

dynamic semantics

**Static semantics.**

- If all the following hold then `(let p = e1 in e2) : t2`:
  - `e1 : t1`
  - the pattern variables in `p` are `x1..xn`
  - `e2 : t2` under the assumption that for all `i` in `1..n` it holds that `xi : ti`,

static semantics

## **3.5.2 Pattern Matching with Functions**

**Static semantics.**

- Let `x1..xn` be the pattern variables appearing in `p`. If by assuming that `x1 : t1` and `x2 : t2` and ... and `xn : tn`, we can conclude that `p : t` and `e :u`, then `fun p -> e : t -> u`.
- The type checking rule for application is unchanged.

static semantics

**Dynamic semantics.**

- The evaluation rule for anonymous functions is unchanged.
- To evaluate `e0 e1`:
  1. Evaluate `e0` to an anonymous function `fun p -> e`, and evaluate `e1` to value `v1`.
  2. Match `v1` against pattern `p`. If it doesn't match, raise the exception `Match_failure`. Otherwise, if it does match, it produces a set $b$ of bindings.
  3. Substitute those bindings $b$ in `e`, yielding a new expression `e'`.
  4. Evaluate `e'` to a value `v`, which is the result of evaluating `e0 e1`.

dynamic semantics

**3.5.3 Pattern Matching Examples:**

Better to see video

**3.6 Type Synonyms**

Type synonym: new name for an already existing type, useful in ways of giving descriptive names to complex types

```
type point = float * float
type vector = float list
type matrix = float list list
```

## 3.7 Options

Options: alpha option, a variant. Can be seen as a wrapped box, its either full or empty.  ‘a option = None I Some of ‘a

```
let get_val default o = match o with
  | None -> default
  | Some x -> x
```

what default value programmer want if there’s nothing in the box, and return alpha

```
utop # Some 45;;
- : int option = Some 45
```

Create an option that is like a box with 45 in it

```
utop # None;;
- : 'a option = None
```

Create an option that is like an empty box

```
utop # let extract p =
        match p with
            | None -> ""
            | Some i -> string_of_int i;;
val extract : int option -> string = <fun>
-( 21:41:21 )-< command 2 >------------------
utop # extract None;;
- : string = ""
-( 21:41:44 )-< command 3 >------------------
utop # extract (Some 32);;
- : string = "32"
```

pattern matching with option value. If parameter p holds a number, turn it to a string and return the string. If p=None, return “”

## 3.8 Association Lists

Map: a data structure that maps keys to values. Easy implementation of a map is a list of pairs

```
utop # let map = [("NY", "New York"); ("MN", "Minnesota"); ("WI", "Wisconsin")];;
val map : (string * string) list =
  [("NY", "New York"); ("MN", "Minnesota"); ("WI", "Wisconsin")]
```

state abbreviation and state names

```ocaml
(** [insert k v lst] is an association list that binds key [k] to value [v]
    and otherwise is the same as [lst] *)
let insert k v lst = (k, v) :: lst

(** [lookup k lst] is [Some v] if association list [lst] binds key [k] to
    value [v]; and is [None] if [lst] does not bind [k]. *)
let rec lookup k = function
| [] -> None
| (k', v) :: t -> if k = k' then Some v else lookup k t
```

insert function(insertion sort), lookup function

## 3.9 Algebraic Data Types

### 3.9.1 Variants that carry data

```ocaml
type point = float * float

type shape =
  | Circle of {center : point; radius: float}
  | Rectangle of {width: point; upper_right: float}

let c1 = Circle {center=(1.2, 3.2); radius=24.2}
let r1 = Rectangle {width=(1.2, 5.3); upper_right=3.2}
```

build variants with passed in data. The of keyword allows data to be passed into the variant type. (Circle/rectangle)

```ocaml
type point = float * float

type shape =
  | Circle of {center : point; radius: float}
  | Rectangle of {lower_left: point; upper_right: point}

let c1 = Circle {center=(1.2, 3.2); radius=24.2}
let r1 = Rectangle {lower_left=(1.2, 5.3); upper_right=(3.2, 6.3)}

let avg a b =
  (a +. b)  /. 2.

let center s =
  match s with
    | Circle {center; radius} -> center
    | Rectangle {lower_left; upper_right} ->
      let (x_ll, y_ll) = lower_left in
      let(x_ll, y_ll ) = upper_right in
      (avg x_ll y_ll, avg x_ll y_ll)
```

pattern matching with objects, in keyword, avg

```
utop # center c1;;
- : point = (1.2, 3.2)
─( 11:09:43 )─< command 2 >
utop # center r1;;
- : point = (4.75, 4.75)
```

center function with passing objects as parameters

## 3.9.2 Syntax and Semantics

Constant: constructor that carries no values
Non-Constant: constructor that carry data
- If e==> v then C e ==> C v, assuming C is non-constant
- If p matches v and produces binding b, then C p matches C v, producing binding b

## 3.9.4 Recursive Variants

OCaml just codes up lists as variants:

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

either [NIL] or cons of alpha a start alpha a list, [] and :: are the constructor for the list

```
type intlist = Nil | Cons of int * intlist

let lst3 = Cons (3, Nil)  (* similar to 3 :: [] or [3]*)
let lst123 = Cons(1, Cons(2, lst3)) (* similar to [1; 2; 3] *)

let rec sum (l : intlist) : int=
  match l with
  | Nil -> 0
  | Cons (h, t) -> h + sum t

let rec length : intlist -> int = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty : intlist -> bool = function
  | Nil -> true
  | Cons _ -> false
```

variant type used to represent something similar to intlist

**<u>3.9.5 Parameterized Variants</u>**

Polymorphism: “poly”(many) and “morph”(form), ‘a is a OCaml feature of parametric polymorphism. The function doesn’t care what the ‘a is in a, and it’s willing to work with any datatypes of a .

```
let rec length : 'a mylist -> int = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty : 'a mylist -> bool = function
  | Nil -> true
  | Cons _ -> false
```

it doesn't matter if the 'a is an int mylist or string mylist, or any (whatever) mylist, it will work for any and all

```
let rec length = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty = function
  | Nil -> true
  | Cons _ -> false
```

type annotation aren't essential, could be omitted. This function will work just like the above functions

```
type ('a, 'b) pair = {first : 'a; second : 'b}
let x = {first = 2; second = "hello"}
```

```
type ('a, 'b) pair = { first : 'a; second : 'b; }
```

possible to have multiple type parameters for a parameterized type, in which case the parentheses are needed  This type is able to have an int and string parameter as datatypes

**3.9.6 Polymorphic Variants**

```
match f 3 with
  | `NegInfinity -> "negative infinity"
  | `Finite n -> "finite"
  | `Infinity -> "infinite"
```

```
- : string = "finite"
```

polymorphic variants starts with a backquote(`) character

**3.10. Exceptions**

OCaml has an exception syntax defined like this: exception E of t , E=constructor name, t=type

```
exception A
exception B
exception Code of int
exception Details of string
```

The of t is optional

```
Failure "something went wrong"
```

create an exception value, the exception value of this constructor is Failure, which carries a string of “something went wrong”

```
raise e
```

to raise an exception value e, write like this

There is a convenient function `failwith : string -> 'a` in the standard library that raises `Failure`. That is, `failwith s` is equivalent to `raise (Failure s)`.

convenient function of fail with in the standard library that raises Failure. failwith s is same as raise ( Failure s)

```
let division x y =
  try x/y with
    | division_by_zero -> 0
```

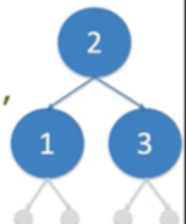
```
utop # division 3 0;;
- : int = 0
-( 20:48:23 )-< comma
utop # division 0 3;;
- : int = 0
```

try parameter with is the same as match parameter with, function returns 0 if it fits the division_by_zero

## 3.11 Example: Trees

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

let t =
  Node (2,
        Node (1, Leaf, Leaf),
        Node (3, Leaf, Leaf)
  )
```




binary tree example code

```
let rec size = function
  | Leaf -> 0
  | Node (_, l, r) -> 1 + size l + size r
```

function to find the size of the binary tree

```
let rec sum = function
  | Leaf -> 0
  | Node (v, l, r) -> v + sum l + sum r
```

function to find the sum of the binary tree

### 3.11.1 Representation with Tuples

```
type 'a tree =                              type 'a mylist =
  | Leaf                                      | Nil
  | Node of 'a * 'a tree * 'a tree            | Cons of 'a * 'a mylist
```

comparison between tree and list, cons carry one sublist, while node carries two subtrees.

```
(* the code below constructs this tree:
         4
       /   \
      2     5
     / \   / \
    1   3 6   7
*)
let t =
  Node(4,
    Node(2,
      Node(1, Leaf, Leaf),
      Node(3, Leaf, Leaf)
    ),
    Node(5,
      Node(6, Leaf, Leaf),
      Node(7, Leaf, Leaf)
    )
  )
```

constructing a small binary tree example

### 3.11.2 Representation with Records

```
type 'a tree =
  | Leaf
  | Node of 'a node

and 'a node = {
  value: 'a;
  left: 'a tree;
  right: 'a tree
}
```

```
(* represents
      2
     / \
    1   3  *)
let t =
  Node {
    value = 2;
    left = Node {value = 1; left = Leaf; right = Leaf};
    right = Node {value = 3; left = Leaf; right = Leaf}
  }
```

example tree with records

```
(** [mem x t] is whether [x] is a value at some node in tree [t]. *)
let rec mem x = function
  | Leaf -> false
  | Node {value; left; right} -> value = x || mem x left || mem x right
```

recursive search over the tree, recursively traversing tree

```
let rec preorder = function
  | Leaf -> []
  | Node {value; left; right} -> [value] @ preorder left @ preorder right
```

function to preorder traversal of a tree

```
let preorder_lin t =
  let rec pre_acc acc = function
    | Leaf -> acc
    | Node {value; left; right} -> value :: (pre_acc (pre_acc acc right) left)
  in pre_acc [] t
```

extra argument acc to accumulate the values at each node, making it linear time.

## <u>3.12 Example: Natural Numbers</u>

Natural number: either zero or the successor of some other natural number, leads to OCaml's type nat

`type nat = Zero | Succ of nat` new type nat, and Zero and Succ are constructors for value for this type

```ocaml
let zero = Zero
let one = Succ zero
let two = Succ one
let three = Succ two
let four = Succ three
```
example of natural number values

```ocaml
let iszero = function
  | Zero -> true
  | Succ _ -> false

let pred = function
  | Zero -> failwith "pred Zero is undefined"
  | Succ m -> m
```

```ocaml
let rec add n1 n2 =
  match n1 with
  | Zero -> n2
  | Succ pred_n -> add pred_n (Succ n2)
```
add two numbers

```ocaml
let rec int_of_nat = function
  | Zero -> 0
  | Succ m -> 1 + int_of_nat m

let rec nat_of_int = function
  | i when i = 0 -> Zero
  | i when i > 0 -> Succ (nat_of_int (i - 1))
  | _ -> failwith "nat_of_int is undefined on negative ints"
```
convert nat value to type in and vice-versa

```ocaml
let rec even = function Zero -> true | Succ m -> odd m
and odd = function Zero -> false | Succ m -> even m
```
determine whether a natural number is even or odd

**Best Practice:**

- quit utop then restart the file with #use “fileName.ml”. Its considered better “hygiene”, prevents old definitions confusing Utop with new definitions