1 - Usando o interpretador *ocaml*, introduza sucessivamente as expressões que se seguem. Interprete os resultados que for obtendo. Para sair faça quit ;;

a) Elementar

```
# 1+2 ;;
# let f x = x + 1 ;;
# f 2 ;;
# let f = (fun x -> x + 1) ;;
# f 2 ;;
# (fun x -> x + 1) 2 ;;
```

b) Recursividade

```
# let fact x = if x = 0 then 1 else x
* fact (x-1) ;;
# fact 5 ;;
# let rec fact x = if x = 0 then 1
else x * fact (x-1) ;;
# fact 5 ;;
```

c) tipos

```
# let f x = x ^ x ;;
# let f (x,y) = x + y ;;
# let f (x,y) = x +. y ;;
# let f (x,y) = (y, x) ;;
# let f (x,y) = x = y ;;
# let g x y = x+y;;
```

2 - Usando um editor de texto, escreva a função fact num ficheiro de texto chamado **fact.ml**. Depois, já dentro do interpretador *ocaml*, use a directiva

```
#  #use "fact.ml" ;;
```

e experimente a função

```
let rec fact x = if x=0 then 1 else x*fact (x-1);;
```

3 - Escreva em ML uma função chamada parque que calcule o preço a pagar no parque de estacionamento subterrâneo dos Restauradores, em Lisboa. A função parque tem 4 argumentos inteiros: hora de entrada, minuto de entrada, hora de saída, minuto de saída. Em Março de 1999, a tabela de preços era a seguinte:

1ª Hora	120 cêntimos
2ª Hora	140 cêntimos
3ª Hora	150 cêntimos
Seguintes	155 cêntimos

Mais informação:

- As horas acumulam. Por exemplo, um carro que esteja no parque 2 horas paga 120+140 cêntimos.
- Os tempos de permanência são arredondados para horas completas e sempre para cima. Por exemplo um carro que permaneça no parque uma hora e um minuto paga duas horas completas.
- Assuma que o carro nunca está no parque mais do que 24:00. Se o momento de entrada for, por exemplo, 12:30 e o momento de saída 10:31, então há a pagar 23 horas (o custo é 3510 cêntimos).

```
let conta he me hs ms = if hs>he && ms>me then hs-he+1 else if hs>he then hs-he else if hs<he &&
ms>me then 24-he+hs+1 else 24-he+hs;;

let parque he me hs ms = let horas = conta he me hs ms in if horas=1 then 120 else if horas=2 then 260
else if horas=3 then 410  else 410 + (horas-3)*155;;
```

4 - Sejam a,b,c inteiros, supostamente comprimentos dos lados dum triângulo. Escreva uma função tri que devolva:

0 Se a,b,c não definirem um triângulo próprio;
1 Se a,b,c definirem um triângulo equilátero;
2 Se a,b,c definirem um triângulo isósceles;
3 Se a,b,c definirem um triângulo escaleno.

NOTA1 - Este exercício serve para tomar contacto com as particularidades do "*if*" em ML.
NOTA2 - Repare que num triângulo próprio, o comprimento de um lado é sempre inferior à soma do comprimento dos outros dois lados.

```
let tri a b c = if a>(b+c) || b>(a+c) || c>(a+b) then 0 else

if a=b && b=c then 1 else

if a<>b && b<>c && a<>c then 3  else 2;;
```

5 - Usando o interpretador *ocaml*, introduza sucessivamente as expressões que se seguem. Interprete os resultados que for obtendo.

a) let

```
# let n = 4 ;;
# let g x = x + n ;;
# g 0 ;;
# let n = 5 ;;
# g 0 ;;
# n;;
```

b) call-by-value

```
# let rec loop x =  loop x ;;
# (fun x -> 5) (loop 3) ;;
```

6 - Defina uma função polimórfica max que determine o máximo entre dois valores. Depois avalie as seguintes expressões: (max 3 7), (max 4.5 1.2), (max "ola" "ole").

```
let max a b = if a>b then a else b;;
```

7 - Defina uma função para determinar o máximo divisor comum entre dois inteiros positivos. Utilize o algoritmo de Euclides.

mdc: int -> int -> int

```
let rec mdc a b = let r = a mod b in

if r = 0 then b else mdc b r;;
```

8 - Programe as seguintes funções sobre listas:

len: 'a list -> int
sum: int list -> int
concat: string list -> string

```
let rec len l = match l with  [] -> 0   | x::xs -> 1 + len xs  ;;

let rec sum l = match l with [] -> 0 | x::xs -> x+ sum xs;;

let rec concat l = match l with []-> "" | x::xs -> x^concat xs;;
```

Exemplos:

len [1;1;2;3;4;1] = 6
sum [1;1;2;3;4;1] = 12
concat ["ola"; ""; "ole"] = "olaole"

9 - Escreva em OCaml uma função

succAll : int list -> int list

que produza a lista dos sucessores duma qualquer lista de inteiros.

Exemplos:
succAll [] = []
succAll [3; 6; 1; 0; -4] = [4; 7; 2; 1; -3]

```
let rec suc l = match l with [] -> [] | x::xs -> (x+1)::suc xs;;
```

10 - Implemente o tipo de dados conjunto. Para representar os conjuntos, use listas não ordenadas mas sem repetições. As funções pretendidas são as seguintes:

```
belongs: 'a -> 'a list -> bool       (* teste de pertença *)
union: 'a list -> 'a list -> 'a list (* união de conjuntos *)
inter: 'a list -> 'a list -> 'a list (* intersecção de conjuntos *)
diff: 'a list -> 'a list -> 'a list  (* diferença de conjuntos *)
```

```
let rec pertence a l = match l with [] -> false | x::xs -> if a=x then true else pertence a xs;;

let rec uniao a b = match a with [] -> b | x::xs-> if pertence x b then uniao xs b else x::uniao xs b;;

let rec inter a b = match a with [] -> [] | x::xs-> if pertence x b then x::inter xs b else inter xs b;;

let rec diff a b = match a with [] -> [] | x::xs-> if pertence x b then diff xs b else x::diff xs b;;
```

Exemplos:
belongs 4 [1;2;3;4;5;6] = true
belongs 4 [1;2] = false
union [7;3;9] [2;1;9] = [7;3;2;1;9]
inter [7;3;9] [2;1;9] = [9]
diff [7;3;9] [2;1;9] = [7;3]
power [1;2] = [[]; [1]; [2]; [1;2]]
power [1;2,3] = [[]; [1]; [2]; [3]; [1;2]; [1;3]; [2;3]; [1;2;3]] (* não interessa a ordem dos elementos *)

11 - Considere o tipo árvore binária em OCaml:

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

Escreva uma função treeToList que transforme uma árvore binária numa lista contendo os mesmos valores. A arrumação dos elementos na lista resultado fica ao seu critério.

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

```
let rec conv t = match t with Nil -> [] | Node(x,l,r) -> x::conv l@ conv r;;
```

12 - Programe uma função:

balanced: 'a tree -> bool

que determine se uma árvore binária está ou não equilibrada. Uma árvore binária diz-se *equilibrada* se, para cada nó, a diferença de profundidades das suas subárvores não superar a unidade. [Terá também de programar a função auxiliar height.]

Exemplos:

```
balanced Nil = true
balanced (Node(3,Node(5,Nil,Nil),Node(6,Nil,Nil))) = true
balanced (Node(1,Nil,Node(2,Nil,Node(3,Nil,Nil)))) = false
```

```
let rec height t = match t with  Nil -> 0  | Node(x,l,r) ->  1 + max (height l) (height r) ;;

let rec balanced t = match t with Nil -> true | Node(x,l,r) -> if (height l) - (height r) > 1 || (height r) - (height l) > 1 then false else true;;
```

13 - Programe uma função:

subtrees: 'a tree -> 'a tree list

que produza a lista de todas as subarvores distintas que ocorrem na árvore argumento.

Exemplos:

```
subtrees (Node(5,Nil,Node(6,Nil,Nil))) =
        [Node(5,Nil,Node(6,Nil,Nil)); Node(6,Nil,Nil); Nil]
subtrees Nil = [Nil]
```

```
let rec sub t = match t with Nil ->[Nil] | Node(x,l,r) -> if l<> Nil && r<> Nil then Node(x,l,r):: sub l @ sub r

else if l=Nil && r<>Nil then Node(x,l,r)::sub r

else if l<> Nil && r=Nil then Node(x,l,r)::sub l

else [Node(x,l,r);Nil];;
```

14 - Primavera. Programe uma função:

spring: 'a -> 'a tree -> 'a tree

que faça crescer novas folhas em todos os pontos terminais duma árvore. As novas folhas são todas iguais entre si.

```
let rec spring t = match t with Nil -> Node(1,Nil,Nil) | Node(x,l,r) -> if l=Nil && r=Nil then
Node(x,Node(1,Nil,Nil),Node(1,Nil,Nil)) else if l=Nil && r<>Nil then Node(x,Node(1,Nil,Nil),spring r)
  else if l<>Nil && r=Nil then Node(x,spring l, Node(1,Nil,Nil)) else Node(x,spring l, spring r);;
```

15 - Outono. Programe uma função:

fall: 'a tree -> 'a tree

que elimine todas as folhas existentes duma árvore. Os nós interiores que ficam a descoberto tornam-se folhas, claro, as quais já não são para eliminar.

```
let rec fall t =  match t with Nil->Nil | Node(x,l,r) -> if l=Nil && r=Nil then Nil else Node(x,fall l, fall r);;
```

16 - Escreva em OCaml uma função:

split : int -> 'a list -> 'a list * 'a list

que particione uma lista dada em dois segmentos. O local de partição é indicado através um número inteiro situado entre 0 e o comprimento da lista. A sua função deve respeitar os seguintes exemplos:

split 0 [1;2;3;4;5] = ([], [1;2;3;4;5])
split 3 [1;2;3;4;5] = ([1;2;3], [4;5])
split 5 [1;2;3;4;5] = ([1;2;3;4;5], [])
split 0 [] = ([], [])

```
let rec split n l = match l with []->([],[]) | x::xs-> let (a,b) = split (n-1) xs in if n>0 then (x::a,b) else (a,x::b);;
```

17 - Escreva em OCaml uma função:

pack : 'a list -> ('a * int) list

para compactar listas, substituindo cada subsequência de valores repetidos por um par ordenado que indica qual o valor que se repete e qual o comprimento da subsequência. Dois exemplos:

pack [] = []
pack [10.1; 10.1; 10.1; 10.1; 10.1; 10.1; 10.1; 10.0; 10.0; 10.1; 10.0] =
[(10.1, 7); (10.0, 2); (10.1, 1); (10.0,1)]

```
let rec divide l = match l with [] -> []  | [x] -> [[x]]  | x::xs -> let (a::y)::ys = divide xs in if x = a then (x::a::y)::ys
else [x]::(a::y)::ys;;
let rec length l = match l with [] -> 0 | x::xs-> 1 + length xs;;
let rec conta t = match t with []->[] | x::xs-> (x,length x)::conta xs;;
let rec pares t = match t with [] -> [] | (a::b,c)::d -> (a,c)::pares d;;
let rec pack t = match t with []->[] | l -> pares(conta(divide(l)));;
```

18 - Escreva agora uma função para descompactar listas.

unpack : ('a * int) list -> 'a list

Dois exemplos:

unpack [] = []
unpack [(10.1, 7); (10.0, 2); (10.1, 1); (10.0,1)] =
[10.1; 10.1; 10.1; 10.1; 10.1; 10.1; 10.1; 10.0; 10.0; 10.1; 10.0]

```
let rec unpack l = match l with [] -> [] | (a,b)::xs -> if b=1 then a:: unpack xs else a::unpack ((a,b-1)::xs);;
```