Cours de Programmation

CAML

Cours écrit sur ordinateur par Sueur Christian.
E-mail : Sueur.christian@gmail.com

Sommaire :

Chapitre 0. Introduction :

Définition :

- Un algorithme c'est une suite d'instruction qui une fois exécutée correctement conduit à un résultat.
- Un algorithme doit contenir uniquement des instructions compréhensibles pour celui qui l'exécute (homme ou machine).

Dans ce cours on va apprendre à écrire un algorithme, et à le transcrire dans un de programmation : CAML (langage fonctionnel different du C qui est un langage impératif).

Programme informatique : c'est un ensemble d'instruction (électronique) exécuté par un ordinateur.

Comme le langage de différent types de machine (mac, pc, wii) est différent on utilise un langage de programmation indépendant de la machine ciblée, et on écrit les instruction tout simplement dans un fichier texte que l'ont transforme en langage machine : *La Compilation.*

Pourquoi CAML ???

CAML est un langage fonctionnel : tout peut être vu au sens des fonctions mathématiques.

1 : CAML fait beaucoup de vérification automatiques et laisse peut de place à l'erreur.
2 : Langage interpréter : il existe un programme (OCAML) qui permet de tester les instructions sans passer par la compilation.

I. Les Fonctions.

I.1 : les bases du langage.

I.1.1 : Les mots-clefs.

Un langage suppose qu'à partir d'un nombre relativement réduit de mots on puisse les combiner entre eux pour former des phrases (instructions). On peut inventer des nouveaux mots dont le sens est figé : les mots clefs

En CAML: Let, if, then, else, function, rec, etc..

Opérateur arithmétique.	Opérateur Booléens.
+ , - , / , *	and, or, not
Opérateur de comparaison.	Flottant.
<> , < , > , <= , >= , =	+. , - . , /. , *.

I.1.2 : Les types de base.

En langage, il existe des mots (objets) de différents types que l'on peut combiner avec des règles spécifiques. (Grammaire, Verbes, Sujet, Adjectif, etc..).

En CAML les types sont de base :

Int	Nombres entiers (1,2,-5,123) -> (+ , - , / , *)
Float	Nombres décimaux (1.4, 2.0) -> (+. , - . , /. , *.)
String	Chaîne de caractère ("Bonjours")
Bool	Booléens (true, false) -> (and, or, not,&)
Functions	Fonctions caractérisées par un type de départ et un type d'arriver (int -> int)
Unit	Type pour les entrées/sorties (toute les actions qui savent du code du langage)

Pour les opérateurs de comparaisons le type peut être entier ou flottant mais les deux valeurs à comparer doivent avoir le même type.

I.1.3 : Les expressions.

Combiner des mots (ou objets) en CAML forme des expressions. Chaque expression est caractérisée par un type. Ce type sera du type du résultat de l'expression.

```
# 2+(3*4) ;;
- :int = 14
```

I.1.4 : Les définitions.

I.1.4.1 : Définitions globales.

En CAML on peut définir des nouveaux objets : une définition.
On utilise pour cela le mot-clef : Let

```
# let x = 2 ;;          (x est une constante du sens mathématique on
Val x :int = 2          l'appel une variable en CAML)
```

```
# let x = 2 = 3 ;;
Val x : bool = false
```

I.1.4.2 : Définition locales.

Définir une variable juste le temps d'une expression. Une fois l'expression évalué la variable retourne à son état initial : soit la variable possède la valeur de la dernière définition globale soit elle n'a pas de valeur.

Le mot-clef pour faire une définition locale est : in

```
# let x = 2 ;;              -> (def globale) val x :int = 2
# let x = 5 in x*2 ;;       -> (def local) -:int = 10
# x ;;                      -> -:int = 2
```

I.1.5: Fonctions.

En CAML on identifie une fonction par le fait qu'elle possède des variables d'entré (ou arguments)

```
# let suivant (x) = x+1;;
val suivant :int -> int
# suivant(2) ;;             # suivant ;;
- :int = 3                  - :int -> int = <fun>
```

I.1.6 : Variable muettes.

Ce sont les variables utilisées dans les déclarations de fonctions. Ces variables ne sont pas définies.

I.1.7 : Définition locale des fonctions.

On utilise in comme pour les variables.

```
# let precedent x = x-1 in (precedent 3)*2;;
-:int = 4
# precedent 4;;
 Unbound value precedent
```

```
# let puiss4 x = let carre y = y*y in carre(carre x) ;;
 val puiss4 :int -> int = <fun>
```

I.1.8 : Trois syntaxe pour la définition de fonction.

```
# let suivant x = x+1;;
val suivant : int -> int = <fun>
```

```
# let suivant = function x  -> x+1;;
 val suivant : int->int = <fun>
```

```
# let (suivant :int->int)= function x->x+1 ;;
val suivant : int->int = <fun>
```

I.1.9 : Fonctions à plusieurs variables.

```
# let moyenne x y = (x+y)/2 ;;
 val moyenne : int -> int->int = <fun>
```

. Attention il est impératif de faire attention aux espaces.

2 exemples d'erreurs commises:

```
                                # let x = 2 ;;
                                # let y = 3 ;;
# let moyennexy = (x+y)/2 ;;      # let moyenne xy = (x+y)/2 ;;
 val moyennexy : int = 2 ;;          val moyenne :'a□ int = <fun>
```

I.1.10: Usage des parenthèses.

Le parenthésage implicite de CAML prend en compte les priorités des opérateurs mathématiques mais va agir comme un parenthésage le plus à gauche dans le cadre d'un appel de fonction :

```
# (moyenne 2 3)*5 ;;
```

Ceci est implicite il faut donc faire

```
# moyenne 2 (3*5) ;;
```

Par contre si on fait

```
# moyenne 3*5 2;;
```

Cela va fonctionner car il va prendre 3*5 comme premier argument et 2 comme deuxième argument

I.2 : Conversion de types et concaténation.

I.2.1: Conversions entier -> flottant.

# float_of_int ;; - : int -> float = <fun>	#int_of_float ;; - : float -> int = <fun>

Exemple:

# float_of_int 2 ;; - : float = 2.0	#int_of_float 2.5 ;; - : int = 2

Si on ne donne pas en entée un flottant qui correspond à une valeur entière, alors la fonction int_of_float renvoie la valeur entière (tronqué) du flottant.

```
# let moyenne x y = (x+y)/2 ;;
#let moyenne x y = ((float_of_int x)+.(float_of_int y))/.2.0;;
Val moyenne : int -> int -> float
```

I.2.2: Conversion d'une chaîne de caractère -> nombre (A Proscrire).

-- Conversion d'un nombre en chaîne de caractères. -*-*

# int_of_string ;; - : string -> int	# float_of_string;; -: string -> float

Exemple:

# int_of_string "34" ;; - : string = 34	# float_of_string "34.0";; -: string 34.0

-- Conversion d'une chaîne de caractères en nombre. -*-*

# string_of_int ;; - : int -> string	# string_of_int;; -: float -> string

Exemple:

# string_of_int 2;; - : string = "2"	# string_of_int 3.8 ;; -: float -> "3.8"

I.2.3: Concaténation.

La concaténation permet de « coller » ensemble 2 chaînes de caractère.
En CAML : ^ (accent circonflexe, chapeau chinois).

# "bon"^"jour";; -: string = "bonjour"	Entre les "" on peut écrire tous les caractères du clavier (sauf ").

I.3 : Polymorphisme.

En informatique, le polymorphisme sert à désigner des objets qui peuvent avoir un type ou des fonctions différentes selon le texte. Les fonctions qui n'acceptent qu'un seul type d'objet sont appelées monomorphe.

En CAML on va exprimer le polymorphisme par un type spécifique : 'a

```
# let fonctionquisertarien x = 0;;
Val fonctionquisertarien : 'a -> int = <fun>
```

# let id x = x;; Val id: 'a -> 'a =<fun>	# let premier a b = a ;; Val premier : 'a->'b->'a =<fun>

"abc"< "bac" -> la comparaison entre les chaînes de caractère suit l'ordre lexicographique

I.4 :Pré-condition, post-condition.

```
# let moy a b = (a+b)/2 ;;
val moy : int -> int -> int = <fun>
# let moy a b = (a+.b)/.2. ;;
val moy : float -> float -> float = <fun>
# let moy a b = float_of_int(a+b)/.2. ;;
val moy : int -> int -> float = <fun>
```

Les pré-conditions et post-conditions servent à décrire le contexte de "justesse" («correction ») d'une fonction.

Pré-conditions : Condition qui doivent vérifier les arguments de la fonction.

Pot-conditions : Condition sur le résultat de la fonction.

I.5 : Quelques fonctions prédéfinis.

En CAML il existe un certains nombres de fonctions prédéfinies

Cos, Sin, Tan (float -> float)
Acos, Asin, Atan (float -> float)
Sqrt: Square Root (Racine carré) (float -> float)

Exemple:

```
# let f x = (cos x)*.(cos x )+.(sin x )*.(sin x);;
val f : float -> float = <fun>
```

Mod: le module donne le reste de la division entière
Int -> int -> int 5 mod 2 = 1.

II. Les conditionnels

Si un nombre est divisible par 3 alors oui sinon non.
Si alors sinon

Jusqu' ici, les taches à réaliser ont fait l'objet de fonctions de plus ou moins complexe, avec un ou plusieurs arguments, mais tous ces arguments passaient par la même chaîne d'argument. Mais ont peut vouloir changer ce traitement en fonction de la valeur des arguments : c'est ce que l'on appelle la prise de décision. De manière général, en programmation, la plupart des décisions sont prise par des structures du type *Si alors sinon*.
Il existe d'autres structures décisionnelles, mais la plupart du temps on peut ce ramener a une structure *Si alors sinon* ou à un ensemble de structure *Si alors sinon* imbriqués.
En CAML: *If Then Else*.

II. 1 : Construction if then else.

If (expression booléenne) Then première expression Else deuxième expression.
La décision sera prise en fonction de la valeur que prend l'expression booléenne qui ce trouve après le If.
Si expression booléenne vaut true alors on va dans le then : la valeur *If Then Else* sera celle de la première expression.
Si expression booléenne vaut false alors on va dans le else : la valeur *If Then Else* sera celle de la deuxième expression.

L'expression qui suit if est toujours de type bool :

```
# if 10 then "Bonjours" else "Bye";;
This expression has type int but is here used with type bool
```

Le type de la première expression (après le then) et de la deuxième expression (après le else) est forcement le même.

```
# if true then 2 else 3.0 ;;
This expression has type float but is here used with type int
```

Si on ne veut rien écrire en else on doit mettre des parenthèses vides sinon le résultat est :

```
# if true then 2 ;;
This expression has type int but is here used with type unit
```

Il faut toujours écrire explicitement le else, il y a toujours autant de then et de esle que de if (pour une expression donnée), et inversement.

Le type de l'expression (if (cond) then exp1 else exp2) et le même que celui de exp1 et exp2.

II. 2 : Le filtrage (ou pattern-matching).

Toute expression utilisant le filtrage peut très bien être réécrite uniquement avec des structures if then esle. Le filtrage sert à rendre plus lisible une imbrication de if then else. Qui dit plus lisible dit plus facile à corriger, débuggé, à vérifier.

```
# let f = function x-> x+1;;
# let f = function arg1-> exp1
                  | arg2-> exp2;;
```

Correspond à :

```
# let x = if (x = arg1) then
exp1 else exp2
```

```
# let f = function arg1 -> exp1
                  | arg2 -> exp2
                  | arg3 -> exp3
                  | arg4 -> exp4;;
```

Peut ce traduire par:

```
# let f x = if (x = arg1) then exp1 else
            if (x = arg2) then exp2 else
            if (x = arg3) then exp3 else exp4;;
```

II. 3 : Gérer les cas d'erreur.

L'objectif et de prévenir automatiquement l'utilisateur que l'argument qu'il vient de rentrer à une fonction provoque une erreur (vérifier automatiquement les pré-conditions).

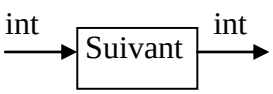
En CAML :
If (l'argument vérifie les pré-conditions) then (j'exécute ma fonction avec cet argument)
else (message d'erreur)

```
# failwith;;
String -> 'a
```

```
# let division a b = if (b<>0)then a/b else failwith "division par
0";;
```

III. Les types composés.

Jusqu´a présent toutes les fonctions rendaient un seul objet en résultat. On peut avoir un ou plusieurs arguments en entrée mais jamais qu´un objet en résultat. On peut représenter une fonction par une « boite ». Les flèches entrantes pour chacun des arguments de la fonction et une flèche sortante pour le résultat.



III. 1 : Types produit.

III.1.1: Définir des m-uplets.

En CAML, on définit des m-uplets avec la virgule «, »

Par exemple :

```
# 2,5 ;;
-: int *int = (2,5)
# "bonjours","tchao" ;;
-: string * string = ("bonjours","tchao")
# 2.0 , 5.8 , 3.6 ;;
-:float* float *float = (2. , 5.8 , 3.6)
```

III.1.2: La virgule est le point.

A ne surtout pas confondre : l´expression où on remplace des virgules par des points (ou inversement) peut syntaxiquement être correctes, mais le sens de l´expression sera complètement différent.

```
# let x = 1, (0 + 3), 5 ;;
val x : int * int * int = (1, 3, 5)
```

III.1.3: Règle de typage sur la « , » et le type produit.

On peut avoir n'importe quel type à gauche et a droite de la virgule ET surtout les types peuvent être différent.

```
# let x = “bonjours”, 2 , 4.5 ;;
val x : string * int * float = ("bonjours", 2, 4.5)
```

Exemple 1 :

Si on veut calculer dans le plan, le milieu de 2 points a et b

```
# let milieu (ax,ay)(bx,by) = (ax +. bx)/.2. , (ay +. by)/. 2. ;;
val milieu : float * float -> float * float -> float * float = <fun>
```

Exemple 2 :

On veut une fonction capable de simuler un jeté de 2 dés a N faces. On utilise pour cela

```
# Random__int m       (renvoie un entier au hasard dans [0, m-1])
```

Ce qui donne

```
# let lancer2dès m = 1 + random__int m , 1 + random__int m;;
Lancer2dès : int -> int * int
```

III.1.4: Accédé a un élément de type composé.

On suppose qu'on a un type composé de triplet de flottant pour représenter les points dans l'espace. Si on définit une variable de ce type

```
# let p = (1.0 , 2.0 , 3.8) ;;
val p : float * float * float = (1., 2., 3.8)
```

Comment a partir de p (et pas a partir de son expression explicite), accéder aux différents éléments (ou champs) de p. On utilise des fonctions accesseurs ou extracteur. En sachant la taille (donc le nombre de champs) du m-uplet passé en paramètre, on peut écrire en CAML

```
# let premierElement = function (x,y,z) -> x ;;
val premierElement : 'a * 'b * 'c -> 'a = <fun>
```

```
# premierElement p ;;
- : float = 1
```

```
# let employe = ("marcel", 129, 0388184274) ;;
val employe : string * int * int = ("marcel", 129, 388184274)
```

III. 2 : Définir de nouveau types.

Utiliser des types composé peut être assez fastidieux à la longue. CAML permet donc de rénover un type produit : une définition de type.

III. 2. 1 : Principe.

Comme pour les définitions de fonction, on met simplement un nom sur un "profile" d'objet. Syntaxe en CAML

```
# let nom_du_type = float * float * float ;;
                    Profile du type
```

Exemple:

```
type point = float * float * float ;;
```

Ici l'objet est du type point et à chaque fois que CAML rencontre le mot point, il va le remplacer par un profile float*float*float. On peut donner plusieurs noms a un même profile, et on peut aussi créer un nouveau type à partir d'un type déjà défini auparavant.

```
# type point = float * float * float ;;
#type pression = float;;
# type surface = float;;
```

Un des grands intérêts de la définition de type, c'est l'économie de représentation. Ca permet de manipuler sans trop de mal des "gros" objets. Ca peut permettre de rendre votre code plus compréhensible par quelqu'un d'autre (si les noms définis correspondent à ce que représente le type).

III. 2. 2 : Les variations syntaxiques sur les types.

Cette manière de définir des types est assez "Faible" dans le sens où la vérification syntaxique porte sur le profil du type et pas sur son nom.

Exemple :

```
# type surface = float ;;
# type longueur = float ;;
# type largeur = float ;;
# let (aireRectangle : longueur * largeur -> surface) = function
(x,y) -> x *. y ;;
val aireRectangle : longueur * largeur -> surface = <fun>
# type pression = float ;;
# let (long :longueur) = 2.5 ;;
# let (larg :largeur) = 2.5 ;;
# let (press : pression) = 1.0;;
# aireRectangle (long , press);;
-: surface = 2.5
```

Parce que CAML vérifie la syntaxe en remplaçant directement les noms par leur profile.

III. 3 : Construire correctement un type.

Bien que les vérifications syntaxiques sur les types définis par CAML soient assez faibles, on veut "bien construire" un type de données avec l'ensemble de ses propriétés. On va toujours construire un nouveau type en 3 étapes : ces étapes sont importantes notamment pour des programmes un peu longs (avec beaucoup de fonction), pour nous permettre de modifier le type sans modifier toutes les fonctions qui l'utilisent.

1- : définir le type (type truc = profile).
2- : écrire une fonction constructeur ou générateur, qui permet de créer une instance d'objet de ce type.
   ➔ c'est ici qu'on peut vérifier la validité des pré-conditions
3- : écrire des fonctions dites accesseurs ou sélecteurs qui permettent d'extraire les différents éléments du type composé.
   ➔ On aura autant d'extracteurs qu'il y a d'élément dans le type composé.

Exemple :
Construire le type couleur est définie par 3 composantes rouge, vert, bleu. Chacune des composantes est un entier entre 0 et 255. (on a du noir quand toutes les composantes valent 0 et du blanc quand elles valent 255).

```
1-  :  # type couleur = int * int * int ;;
2-  :  # let (creerCouleur : int -> int -> int -> couleur) =
       function r -> function v -> function b -> if ((r < 0)or(r >
       255)or(v < 0)or(v>255)or (b<0)or(b>255))then
       failwith"les composantes doivent être entre 0 et 255" else
       (r,v,b) ;;
3-  :  # let (rouge : couleur -> int) = function (r,v,b) -> r ;;
       # let (vert : couleur -> int) = function (r,v,b) -> v ;;
       # let (bleu : couleur -> int) = function (r,v,b) -> b ;;
```

Addition de 2 couleurs :

```
# let (addition2couleur : couleur -> couleur -> couleur) =
function c1 -> function c2 -> creerCouleur (rouge c1 + rouge
c2)(vert c1 + vert c2)(bleu c1 + bleu c2) ;;
# let noir = créerCouleur 0 0 0 ;;
# let blanc = créerCouleur 255 255 255 ;;
# addition2couleur noir blanc ;;
- : couleur = (255, 255, 255)
```

IV. La récursivité

IV .1 : Généralités.

Définition : une fonction est définie récursivement lorsque sa définition utilise f elle même.
Comme pour le découpage fonctionnel, on va utiliser la récursivité pour réduire un problème complexe en un problème simple qu'on sait traiter.
Soit un problème P_n qu'on ne sait pas résoudre directement dans tous les cas, si on connaît une solution simple du problème (notons P_0), l'idée est de trouver une transformation de P_n en un problème moins complexe P_{n-1} qui nous rapproche de la solution simple.
=> on fait de même a partir de P_{n-1}, et ainsi des suite jusqu'à arriver a P_0.
De manière générale la structure qu'on va utiliser pour un raisonnement en récursif sera :
Solution (cas_complexe)= si (cas_simple) alors solution_simple sinon traitement (solution(cas_moins_complexe))

Cas terminal condition d'arret Appel récursif

IV.2 : Analyse avec la récurrence.

Une suite d'un ensemble E est une famille d'élément de E, indexée par l'ensemble des entiers naturels ou par une partie de P Є N. On note une suite (Un), et on dit que Un est le terme de rang n de la suite U.

1. Si Un peut être calculé directement (en fonction de n) on dit qu'on connaît le terme général de la suite et donc on peut exprimer la suite analytiquement.
2. Si il existe un entier n_0 tels que $n \geq n_0$, n Є N, on peut définir la suite par une relation de récurrence : le terme de rang n est donné comme fonction de n et des termes de rang ou d'indice k, avec $k \leq n$

Exemple 1 : suite arithmétique de premier élément a et b de raison r :
Terme globale $U_n = a + n.r$
Récurrence $U_0 = a$
$U_n + 1 = U_n + r$

Exemple 2 : suite géométrique de premier élément a et de raison q
Terme globale $Un = a.q^n$
Récurrence $U_0 = a$
$U_{n+1} = q.U_n$

Remarque A : en termes d'informatique il est bien sur plus avantageux d'utiliser le terme général de la suite pour en calculer un terme d'un rang donné (moins d'opération arithmétique à effectuer, on est plus rapide).
Par contre si on veut connaître un ensemble de valeurs de la suite (de terme de rang consécutif) il est souvent plus avantageux d'utiliser la définition par récurrence.
Remarque B : il existe de nombreux cas ou on ne connaît pas le terme général de la suite.
Exemple : Suite de Fibonacci.
Suite des factorielles : $U_n = n!$
$U_0 = 1$
$U_{n+1} = (n+1)U_n$

IV.3 : La récursivité en CAML.

Le mot-clef "rec" indique à l'intercepteur CAML que la fonction que l'on est en train de définir est fonction récursive (donc va devoir s'appeler elle même). Le "rec" vient se placer entre le "let" et le nom de la fonction.

```
# let rec facto n = if (n = 0)then 1 else n * facto (n-1);;
```

Que se passe t'il si on oublis le rec ?
Dans ce cas CAML renvoie une erreur lorsqu'il rencontre l'appel récursif de la fonction définie : unbonned value facto.

Règle de bonne formation.

1 : le corps d'une fonction récursive doit toujours exprimer un choix, soit par une expression conditionnelle soit par une définition par cas (filtrage).
2 : Tous les appels récursifs doivent l'être sur des valeurs plus petites que celles de l'argument.
3 : au moins un cas terminal (ou condition d'arrêt) rend une valeur qui n'utilise pas la fonction définie

En Pratique :

# facto 3 ;; -> 3 * facto 2 -> 3 * 2 * facto1 -> 3 * 2 * 1 * facto -> 3 * 2 * 1 * 1 = 6	L'ensemble de ces opérateur sont stockées en mémoire pour permettre a CAML de "remontrer" la solution : on appelle la pile cette espace mémoire

La taille de la pile (stack en anglais) est limitée. Donc avec un argument de départ très grand(1) ou si on oublis de mettre une condition (2) ou si la condition d'arrêt n'est jamais atteinte(3) alors on aura plus assez de place dans la pile pour stocker les appels récursifs : erreur CAML = stack overflow (débordement de pile)

(1) : Machine n'est pas assez puissante.
(2)-(3) : problème dans votre fonction récursive.

Ici la fonction facto prend pour près conditions que l'argument, n doit etre un entier naturel (*pré : m ≥ 0*)

```
# let factorielle m = if (m < 0) then failwith "m doit appartenir
N" else facto m;;
```

En CAML, il existe un outil pour "tracer" (pour suivre) les appels récursifs.
Fonction prédéfinie # trace. (# a rajouter en plus du # de l'interpréteur)

```
# # trace facto;;
facto is now traced
facto 2;;
facto <- 2
facto <- 1
facto <- 0
facto -> 1
facto -> 1
facto -> 2
-:int = 2
# # untrace facto;;
```

V : Les listes

V.1 : Motivations

On a vu dans les chapitres précèdent, comment écrire en CAML des fonctions qui prennent en entrée un certain nombre de variables plus ou moins complexes (ou au sens du type composé) et de même qui rendent un résultat plus ou moins complexes. La limitation principale vient du fait que le nombre d'objets en entrée doit forcement être connue à la définition. De même on ne peut rendre qu'un seul objet en résultat, aussi composé soit il.

Exemple : Pour une fonction qui prend en entrée l'ensemble des étudiants d'un groupe de TD (23) soit on a 23 variables en entrée, soit on passe m 23 uplets d'objets du type étudiant. Si le nombre d'étudiant change, il faut réécrire toute la fonction :

V.2 : La construction des listes en CAML.

V.2.1 : Principe.

En CAML, les listes sont définies récursivement. Une liste l d'objets de type t est définit par :

- Soit la liste vide.
- Soit l'association d'un élément a de type t avec une autre liste l d'élément de type t. (On dit que a est la tête de la liste l)

Notation CAML

- La liste vide est noté [ ]
- L'association d'un objet a de type t avec la liste l est noté a::l (on lit "a conss l").

Attention, l'opérateur :: n'est pas commutatif, on peut écrire l::a

Pour cette opération à droite on a toujours une liste d'élément du type t, et a gauche un objet du type t. On ne peut avoir que des objets de même type dans une liste. Avec l'opérateur :: et la liste vide [], on peut construire les liste qu'on veut.

V.2.2 : Première liste.

On veut définir un objet comme étant une liste vide :

```
# let vide = [] ;;
Val vide : 'a list = []
```

On ajoute l'entier 9 à cette liste :

```
# let l1 = 9 ::vide ;;   (*-  égale a #let l1 = 9 ::[] ;;  -*)
Val l1 : int list = [9]
```

Si on veut ajouter plusieurs éléments dans une liste on peut enchaîner ses action :

```
# let l05 = (0 ::(1 ::(2 ::(3 ::(4 ::(5 ::[])))))) ;;
Val l05= int list = [0;1;2;3;4;5]
# let l = [0;1;2;3;4;5];;
Val l = int list = [0;1;2;3;4;5]
```

V.2.3: erreur de type.

1:

```
# let f x = 0 :: x;;
Val f : int list -> int list = <fun>
```

::x => x est une liste ‘a list
0 :: => type d’objet dans la liste int list

2 :

```
# let x = [0 ::l1] ;;
Val x = int list list.
```

V.3: algorithme de base sur les listes.

On va voir les algorithmes qui permettent de manipuler des listes et écrire en CAML les fonctions équivalentes

V.3.1 : création de la liste vide.

```
# let vide = [] ;;
Val vide : ‘a list = []
```

V.3. 2 : Ajouter un élément en tête de liste

```
# let add a l = a ::l ;;
Val  add : ‘a -> ‘a  list =<fun>
```

V.3. 3 : détecter si une liste est vide.

```
# let est_vide = function [] -> true
                          |_-> false.
```

V.3. 4 : extraire le premier élément d’une liste

```
# let tete = function []-> failwith "pas de tete"
                      | t :: r -> t ;;
Val tete : ‘a list -> ‘a = <fun>
```

V.3. 5 :Reste de la liste.

```
# let reste = function [] -> failwith "pas de reste"
                       | t :: r -> r ;;
Val reste : ‘a list -> ‘a list = <fun>
```

V.3. 6 : longueur d'une liste.

```
# let rec longueur = function []-> 0
                                  | t ::r -> 1+ longueur r ;;
# let rec longueur l = if |(l=[])|then 0 else 1+ longueur (reste l);;
                          |(est-vide)|
```

V.3.7 : la somme des élément d'une liste.

Exemple : liste d'entier ou de flottants
ENTIER

```
# let rec som_int = function []->0
                             |t ::r ->t + som_int r;;
# let rec som_float = function [] ->0.
                               |t::r -> t+. som_float r ;;
```

V.3.8: l'égalité entre deux listes.

```
# let l1 = [1 ;2 ;3] ;;
# let l2 = 1 ::2 ::3 ::[] ;;
# l1 = l2 ;;
- : bool = true
# l1 = l3 ;;
- :bool = false
```

Exercices : écrire la fonction qui test l'égalité sur les termes de la liste.

Algo ?
Si les listes sont vides, alors elles sont égales.
Si une des deux listes est vide et pas l'autre, alors elles sont différentes.
Si la tête des deux listes est égale alors on regarde la tête du reste sinon les listes sont différentes

```
# let est_vide = function [] -> true
                          |_-> false ;;
# let rec sontEgale l1 l2 = if (est_vide l1 & est_vide l2)
                            then true
                            else if ((est_vide l1 & not(est_vide l2))
or not (est_vide l1) & (est_vide l2))
                                 then sontEgale (reste l1)(reste l2)
                                 else false ;;
# let rec egale l1 l2 = match (l1,l2) with ([],[]) -> true
                       |(-,[]) -> false
                       |([], -)-> false
                       |(a::r,b::r)-> if (a=b)
                                      then egal r r1
                                      else false;;
```

V.3.9: Concaténation entre 2 listes.

```
# let l1 = [1;2];;
# let l2 = [3;4];;
# l1 @ l2;;
-: int list = [1;2;3;4]
# l1 @ l1 @ l2;;
-: int list = [1;2;1;2;3;4]
```

Exercice : Écrire une fonction qui fait la même chose mais sans utiliser l'opérateur @

Algo ?

Si l1 est vide alors l2 sinon on ajoute la tête de l1 au résultat de la concaténation du reste de l1 et l2.

```
# let rec conca l1 l2 = if l1 = []
                        then l2
                        Else tete l1 :: conca (reste l1) l2 ;;
Conca l1 l2 -> 1::2::3::[4]
```

V.3.10: Vérifier qu'une liste est croissante.

Algo ? Fonction CAML ?

Algo : Si la liste est vide, on renvoi un message d'erreur.
Si la liste n'a pas qu'un seul élément, alors elle est "bien rangée" (-> true)

```
# let rec croissante = function [] -> failwith "liste vide"
                                | t :: [] -> true
                                | t1 :: t2 :: r -> if (t1<t2)
                                                   then croissante (t2::r)
                                                   else false
```

Ou

```
# let rec croissante = function [] -> failwith "liste vide"
                                | t :: [] -> true
                                | t :: r -> if (t<tete r)
                                            then croissante r
                                            Else false
```

V.3.11 : Appartenance d'un élément a une liste

Algo ? Fonction CAML ?

Si la liste est vide alors false sinon on regarde si la tête est l'élément qu'on cherche alors true sinon on recommence sur le reste

```
# let rec appartient l1 a = if l1 []
                            Then false
                             Else if a = tete l1 then true else
appartient (reste l1) a ;;
```

```
# let rec appartient a = function []-> false
                                  | t ::r -> if (t=a)
                                              Then true
                                               Else appartient a r;;
```

V.3.12: Rendre le i-ème élément d'une liste.

i>0 : la tête de la liste est le premier élément

Algo ? Fonction CAML ?

| Ou si la liste et vide |
Si i <= 0 et | longueur de la liste < i | alors failwith "" sinon si i = 1 alors on renvoi la tête de la liste sinon on recommence sur i-1 et le reste

```
# let rec iemeElement l i =if i< 1 then failwith "mauvais i"
                                  else match l with []
                                ->failwith"liste trop petite"
                                          | t :: r if i=1
                                          then t
                                         else iemeElement (i-1)r ;;
```

V.3.13: Création de liste d'intervalles

1 : Une fonction qui prend des borne inf et sup et renvoient la liste [inf; inf +1; inf + 2 ;…..; sup – 1 ; sup]. (ordre croissant).
2 : De même dans l'ordre décroissant.
3 : Finalement une fonction qui prend 2 bornes quelconques et construit la liste [b1,..., b2]

Algo ? Fonction CAML ?

1: inf et sup ordre croissant
Si inf = sup alors on renvoi [sup] sinon on ajoute inf en tete de la liste construite par l'appel récursif avec inf = i + 1 et sup

Construit 1 4
1 :: (construit 2 4)
2 :: (construit 3 4)
3 ::(construit 4 4)
[4]
(1 ::(2 ::(3 ::[4])))

```
# let rec infsup = if inf = sup
                   then [sup]
                   else inf :: infsup (inf+1) sup ;;
# let rec supinf = if inf = sup
                  then [inf]
                  else sup::supinf (sup-1)inf;;
# let construitListe b1 b2 = if (b1 <= b2)
                             then infsup b1 b2
                             else supinf b1 b2;;
```

V.3.14: Supprimer un élément dans une liste.

Plus précisément on veut supprimer la première occurrence de l'élément m dans la liste
Exemple : [1, 2, 3, 4, 3, 5] 3
Se qui donne -> [1, 2, 4, 3, 5]

Algo ? fonction CAML ?

Si la liste est vide alors on renvoi []
Si la tête de la liste est égale à a alors on renvoi le reste sinon ajouter t en tete de la liste rendue puis recommence sur le reste

```
# let rec supprime a = function [] -> []
                              | t ::r -> if t = a
                                         then r
                                         else t :: (supprime a r);;
# let rec supprime a l = match l with [] ->[]
                        |t ::r -> if t=a
                                  then r
                                  else t :: (supprime a r);;
```

Soient l1 et l2 ne contenant aucun doublons (*pré*)
Ecrire un fonction qui teste si l1 est une permutation de l2
(on a écrit la fonction qui teste l'appartenance d'un élément à une liste précédemment)
App l1 a -> true, false

Algo -> code
Soit on écrit une fonction récursive intermédiaire qui fait ce test et qui est appelle la 1ere fois pour une fonction qui compare la longueur des 2 liste
Soit on supprime la tête de l1 dans l2
-> la condition d'arrêt devient l1 = [] et l2 =[]

Exo sur la queue de liste

1. Fonction qui renvoie la queue d'une liste (le dernier élément) ‘a list -> ‘a
2. fonction qui ajoute un élément en queue de liste ‘a -> ‘a list -> ‘a list
3. fonction qui supprime un élément en queue de liste ‘a list -> ‘a list

```
# let rec renvoie = function [] ->failwith "pas bon c'est vide"
                            |[a]-> a
                            | t ::r -> renvoie r ;;
```

```
# let rec ajoute_queue x l = match l with [] -> [x]
                                          | l :: r ->
                                          t ::(ajoute_queue x r ) ;;
```

```
# let rec supp = function []-> failwith "erreur"
                        | [a]->[]
                        | t :: r -> t supp r;;
```

Passer une function en parameter

On prend une fonction p qui pour chaque objet d'un certain type renvoie un booléen pour indiquer si oui ou non l'objet vérifie le propriété :

```
Ex : let pair x = x mod 2 = 0 ;;
```

1 : écrire une fonction qui prend en entrée une telle propriété et une liste, et qui renvoie le nombre d'élément de la liste vérifiant la propriété
(‘a->bool)-> ‘a list -> int
2 : la même chose mais on veut supprimer de la liste tout les élément qui vérifie la propriété.
(‘a -> bool) -> ‘a list -> ‘a list

```
# let rec cptProp prop l = if l =[]
                          then 0
                          else if prop (tete l )= true
                               then 1 + cptProp prop (reste l)
                               else cptProp prop (reste l);;
```

```
# let rec cptProp prop l = match l with
                         []-> 0
                         | t ::r -> (if (prop t)) then 1 else 0) +
cptProp prop r ;;
```

```
Let rec supprProp prop l = match l with
                           []->[]
                           | t ::r if (prop l)
                                   then supprProp prop r
                                   else t :: (supprProp prop r);;
```

Construction automatique de listes

1 : écrire une fonction qui contient une liste de n élément rentrée ou élevé par l'utilisateur. (Utiliser la fonction read_int), qui attend un nombre entier rentré au clavier et renvoie sa somme

2 : écrire une fonction qui contient une liste de n entier a valeur aléatoire (utiliser la fonction random_int, ex : Random_int 20 -> renvoie un nombre entre 0 et 20)
On passera la taille de la liste et valeur max que peuvent avoir les éléments en paramètre de la fonction
3 : on pourra également faire une fonction qui contient la liste composé des n premier entier rangés dans l'ordre croissant

```
#let rec construire n =if n = 0
                       then []
                      else read_int () :: construire (n-1);;
```

```
#let rec comme_tu_veut n m = if n = 0
                             then []
                             else Random_int m ::
comme_tu_veut (n-1)m ;;
```

Ecrire une fonction qui renvoie le plus petit élément d'une liste

[a ;b ;c ;d]
a < plus petit [b ;c ;d] in
if (a< toto)

```
# let rec pluspetit l = match l with
                        [] -> failwith "pas de plus petit élément "
```