Présentation de OCaml

Option informatique MPSI pour moi

Table des matières

Introduction

Nous utiliserons le langage Caml. Son nom était à l'origine un acronyme pour **C**ategorical **A**bstract **M**achine **L**anguage. C'était une contraction de CAM, la Machine Abstraite Catégorique, et ML, la famille de langages de programmation à laquelle Caml appartient. Le nom Caml est resté à travers l'évolution du langage, bien que l'implantation actuelle n'ait rien à voir avec la CAM. Caml fut d'abord conçu et implanté par l'équipe Formel de l'INRIA, dirigée par Gérard Huet. Son développement continue aujourd'hui dans l'équipe Cristal. Pour en savoir plus : http ://caml.inria.fr/about/history.fr.html

Caml est un langage de programmation généraliste, conçu pour garantir la sûreté et la fiabilité des programmes. Il est très expressif et néanmoins facile d'apprentissage et d'emploi. Caml se prête à la programmation dans un style fonctionnel, impératif ou orienté objets. Il est développé et distribué par l'INRIA depuis 1985.

Nous utiliserons plus précisément le système OCaml (pour Objectif Caml).

Les principales caractéristiques de Caml, sont les suivantes :

— Caml est un langage compilé : on écrit les programmes à l'aide d'un éditeur de textes, et le tout est traité par un compilateur pour créer un exécutable.(Cependant les environnements de programmation récents permettent de l'utiliser comma un langage interprété, où les morceaux de programmes sont traités et exécutés à la volée par le système.) (Cependant tous les compilateurs Caml proposent une boucle d'interaction « toplevel », qui ressemble à s'y méprendre à un interprète. En effet dans le système interactif, l'utilisateur tape des morceaux de programmes (des « phrases » Caml) qui sont traitées instantanément par le système, qui les compile, les exécute et écrit les résultats à la volée).

— Caml est un langage fonctionnel. Les fonctions sont des valeurs à part entière qui peuvent être argument ou valeur d'une fonction.

— Caml est un langage fortement typé. Les types des variables ne sont pas déclarés par le programmeur mais calculés automatiquement par l'interprète Caml.

— Caml supporte le filtrage, le polymorphisme et le traitement des exceptions.

— Caml est adapté à la programmation récursive, mais permet la programmation impérative. (Le programme est décomposé en suite d'instructions que l'ordinateur exécute séquentiellement.)

Malgré ses origines académiques Ocaml est un langage dont l'utilisation dépasse le domaine de la recherche en langages et leurs outils associés. Il est utilisé pour l'enseignement dans des nombreuses universités, pas seulement en France mais aussi aux Etats Unis et au Japon. Il est de plus en plus adopté en industrie : dans l'industrie aéronautique pour l'analyse des programmes, en vertu de sa sûreté de programmation (projet Astrée : Analyse Statique de logiciels Temps-Réel Embarqués) ; pour le contrôle des systèmes critiques en avionique (Airbus) et dans le secteur nucléaire. Des grands groupes de l'industrie du logiciel utilisent Ocaml (F# : Microsoft, Intel, XenSource). Il est employé également pour écrire des applications dans le secteur financier (Jane Street Capital, Lexifi), par des projets libres comme MLDonkey (peer-to-peer), GeneWeb (logiciel de généalogie), Unison (logiciel de synchronisation de fichiers multi-plateforme dans certaines distributions Linux), par certains logiciels de l'environnement KDE, par des systèmes bio-informatiques pour l'analyse des protéines, et par beaucoup d'autres.

Jean Mouric, un collègue de Rennes, maintient à jour un environnement de développement intégré regroupant un éditeur de texte et un interprète de commande pour une utilisation interactive de OCaml .

http ://jean.mouric.pagesperso-orange.fr/

Caml vs Python

Python est un langage de programmation **impérative**. Ce paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme (ensemble des variables du programme). En général cela consiste à partir d'un état initial, exécuter une séquence finie de commandes (d'affectation) modifiant l'état courant. On utilise les boucles `for` et `while` pour les répétitions ...

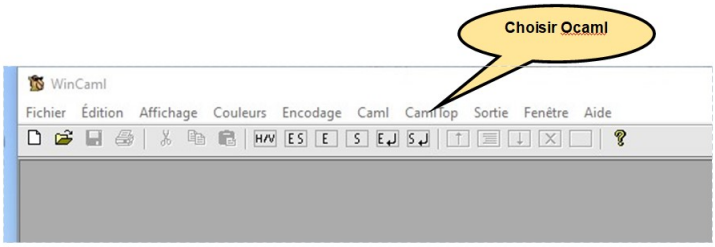
Caml est un langage de programmation **fonctionnelle**. Ce paradigme de programmation considère le calcul comme l'évaluation de fonctions mathématiques. Ainsi le programme est une fonction au sens mathématique, l'exécution étant l'évaluation d'une fonction. On remplace les boucles par l'usage de la récursivité.

En fait Python permet aussi de faire de la récursivité et Caml de la programmation impérative. Les deux langages permettent aussi de faire le paradigme orienté objet.

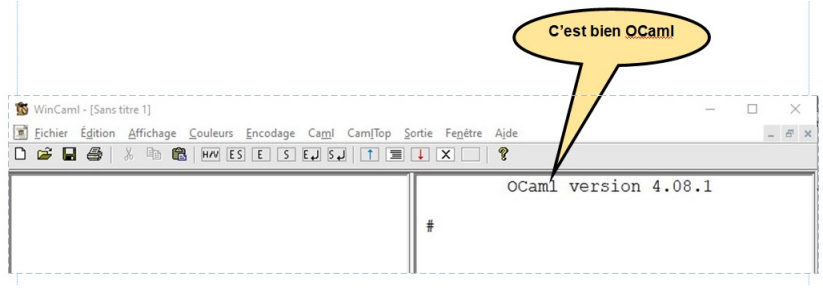
Environnement WinCaml

La version WinCaml installée au lycée est une version portable. Vous pouvez donc copier l'ensemble du répertoire sur une clé USB ou sur votre ordinateur portable et l'utiliser sans installation supplémentaire.

Une fois WinCaml lancé il faut vérifier que l'exécution se fera avec OCaml et non Caml-Light.



On le vérifie en ouvrant un nouveau fichier :



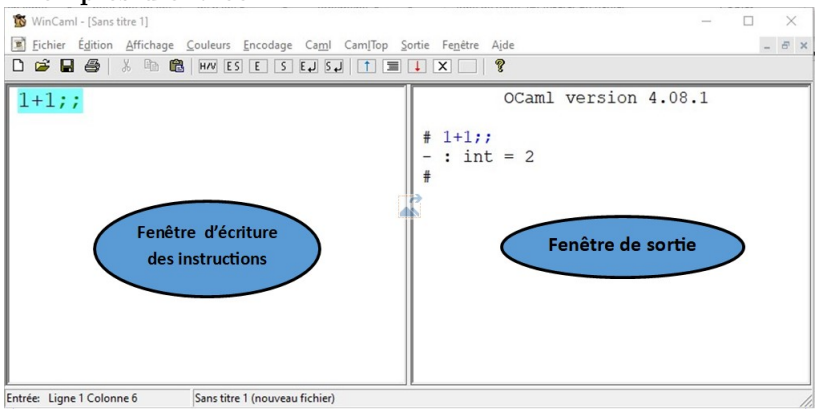
I Calculs et variables

Règle importante : un calcul ou une déclaration de fonctions, de variables, doit toujours se finir par `;;`.

On lance l'exécution (la compilation) à l'aide de ctrl+entrée, ou entrée du pavé numérique.

À l'inverse, les passages à la ligne sont facultatifs (mais recommandés pour faciliter la lecture), ainsi que l'indentation. Cette dernière est à utiliser comme dans le cadre de Python aussi pour une meilleur lisibilité des programmes.

Exemples d'entrée :




```
1 + 1;;
    - : int = 2
```

La première ligne est l'instruction exécutée, tapée dans le shell.
La seconde est la réponse de l'ordinateur (qui apparait dans une fenêtre séparée).
Ici le signe - indique que nous avons simplement calculé une valeur. Cette valeur est de type entier (`int`) et qu'elle vaut 4 .

1) Définitions globales

De même qu'en mathématiques on écrit : « soit s la somme des nombres 1, 2 et 3 »,
on écrit en Caml (« soit » se traduit par let en anglais) :

```
let s = 1 + 2 + 3;;
    val s : int = 6
```

Caml nous répond que nous avons défini un nouveau nom (qu'on appelle aussi variable ou **identificateur**) `s`, qui est de type entier (`: int`) et vaut 6 (`= 6`).

Remarque importante : les identificateurs ne doivent pas commencer par une majuscule ! !

Une fois défini, un nom a toujours la même valeur

Même lorsqu'on définit un identificateur à l'aide d'une expression, une fois celle-ci définie, donc évaluée, elle ne change plus. On dit que **le passage se fait par valeur** :

```
let a = 2;;
    val a : int = 2
let b = a + 2 ;;
    val b : int = 4
let a = 1 ;;
    val a : int = 1
b ;;
    - : int = 4
```

2) Définitions locales

On peut donner une valeur à un nom temporairement, le temps d'un calcul. On parle alors de définition locale :

```
let s = 20 in s * 4;;
     - : int = 80
```

Cette définition est bien locale au sens temporaire car le nom `s` a toujours la valeur définie au début :

```
s;;
    - : int = 6
```

3) Déclaration multiple

```
let x=2 and y=3 ;;
    val x : int = 2
    val y : int = 3
```

ou encore

```
let (x,y) = (2,3);;
    val x : int = 2
    val y : int = 3
```

Attention : cela agit comme une déclaration simultanée. On ne peut donc pas déclarer un second identificateur dépendant du premièr.

```
let z = 2 and t = z+1;;
    Line 2 | characters 18-19:
    2 | let z = 1 and t = z + 1;;
                        ^

    Error: Unbound value z
```

4) Le référencement : comment changer la valeur d'une variable

On écrit parfois en informatique "$k \leftarrow k+1$"

Sans redéfinir la variable k, ce qui serait un peu absurde, c'est impossible en Caml. Il faut déclarer k comme une variable modifiable.

```
let k = ref 0;;
    val k : int ref = {contents = 0}
```

On appelle la valeur à l'aide de " !"

```
!k;;
     - : int = 0
```

Pour changer la valeur référencée on utilise l'affectation, en prenant soin d'appeler la valeur référencée et non l'identificateur.

```
k := !k+1 ;;
     - : unit = ()
k ;;
     - : int ref = {contents = 1}
```

II Les fonctions

1) Déclaration

Les fonctions étant des variables comme les autres dans la programmation fonctionnelle, la déclaration est identique. On définit une fonction en déclarant le résultat d'un appel à cette fonction :

```
let f x = x*2 + 1;;
    val f : int -> int = < fun >
```

Dans ce cas `f` a une valeur qui est du type function (et l'argument `x` n'existe qu'à l'intérieur de la fonction) :

```
f ;;
     - : int -> int = <fun>
```

Pour utiliser une fonction on peut, comme en mathématiques, l'appeler avec des parenthèses, ou sans parenthèses mais en faisant attention de bien comprendre ce qu'on fait.

```
f(3) ;;
     - : int = 7
f 3 ;;
     - : int = 7
f (3*s)
     - : int = 37
f 3*s ;;
     - : int = 42
```

Dans ce dernier exemple `f` est évaluée avec la première valeur rencontrée qui est `3`, puis le résultat est multiplié par 6.

On peut aussi définir une fonction localement à l'intérieur d'un calcul ou de la définition d'une autre fonction :

```
let g x = x*x*x in g 3 * g 4 ;
    - : int = 1728
let g x = let h x = x*x - 1
            in h(2*x)-h(3*x) ;;
    val g : int -> int = <fun>
```

On n'est pas obligé de déclarer une nouvelle fonction locale pour répéter une expression :

```
let g x = let expr = x*x-1 in expr*expr ;;
    val g : int -> int = <fun>
```

2) Fonctions à plusieurs variables

-*Fonction curryfiée*(venant du nom du mathématicien : Haskell Curry)

```
let moyen a b = (a+b)/2;;
    val moyenne : int -> int -> int = <fun>
```

Cette fonction de deux variables reçoit ses arguments un par un. Cela permet d'avoir l'application partielle une fois le `a` donné.

-*Autre méthode :*

on peut définir la fonction avec un seul argument qui est alors un couple.

```
let moyennebis (a,b) =(a+b)/2;;
    moyennebis : int * int -> int = <fun>
```

Il faut dans ce cas donner l'argument complet qui est le couple.

3) Les fonctions anonymes :

On peut créer une fonction sans lui donner de nom. Pour cela on l'introduit par :

— le mot-clé `function` suivi de la formule qui la définit pour une fonction d'une seule variable

— le mot-clé `fun` pour une fonction de plusieurs variables (currifiée)

. C'est utile lorsque l'on manipule des fonctions qui prennent en entrée des fonctions ou qui renvoient des fonctions en sortie.

```
let opp f = (function x -> - f x);;
    val opp : ('a -> int) -> 'a -> int = <fun>
```

```
let g = opp (function x -> 3*x);;
    val g : int -> int = <fun>
let opp2 f = (fun x y -> - f x y);;
    val opp2 : ('a -> 'b -> int) -> 'a -> 'b -> int = <fun>
```

4) Fonctions récursives

Comme on ne peut pas utiliser un identificateur non encore défini dans une déclaration, pour déclarer une fonction qui s'appelle elle même on précise que la fonction est récursive par `rec`.

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

5) Impression

```
print_string "bonjour !";;
     Bonjour !- : unit =()
print_endline "Bonjour !";;
     Bonjour !
 - : unit = ()
```

L'impression s'est produite comme prévu. Cependant Caml nous indique aussi que nous avons calculé un résultat de type unit et qui vaut (). Le type unit est un type prédéfini qui ne contient qu'un seul élément, « () », qui signifie par convention « rien ». Nous n'avons pas demandé ce résultat : tout ce que nous voulions, c'est faire une impression (un effet). Mais toutes les fonctions Caml doivent avoir un argument et rendre un résultat. Lorsqu'une fonction opère uniquement par effets, on dit que cette fonction est une procédure On utilise alors « rien », c'est-à-dire (), en guise de résultat ou d'argument.

La différence entre les deux fonctions utilisées est que la seconde passe à la ligne après l'impression, tandis que la première s'arrête en milieu de ligne.

6) Fonctions mathématiques usuelles

Elles prennent en entrée un flottant, et renvoient un flottant :
`exp, sin, cos, tan, log` (il s'agit bien du logarithme népérien), `sqrt, acos, asin, atan`

III Structures conditionnelles

1) Les opérateurs de comparaisons

Les opérateurs pour les types de base sont : `=`,`<>` ,`<`, `>`, `<=`, `<=`.
On dit que ce sont des opérateurs polymorphes.

2) if . . .then . . .else

```
let minimum x y =
      if x < y then x else y;;

 val minimum : 'a -> 'a -> 'a = <fun>
```

Cette structure conditionnelle apparaît intuitive, mais plusieurs particularités sont à noter :

— La structure `if...then...else...` n'est pas une **instruction**, mais un **objet**. Cet objet est égal soit au contenu du `then`, soit au contenu du `else`.

— L'objet `if...then...else...` doit être d'un certain type, cela impose que les contenus du `then` et du `else` sont de même type. Dans l'exemple ci-dessus, le type n'est pas précisé, mais les deux arguments de la fonction doivent être de même type. (Ce qui suit le `if` est par contre toujours un booléen).

— Il est cependant possible d'exécuter des instructions dans un `if...then...else...`, en utilisant des objets de type unit.

— C'est ce qui se passe si le `else` est omis : l'ordinateur considère alors qu'il contient (), la structure entière est donc de type unit.

— Si le `then` et/ou le `else` contiennent plusieurs instructions, il est alors nécessaire de les délimiter par les mots clés `begin` et `end`.

— Le mot *elif* n'a pas d'équivalent en Caml, il ne peut y avoir que deux cas dans un `if...then...else...`.

— Les passages à la ligne sont facultatifs, on s'adapte selon la longueur des clauses. En cas de passage à la ligne, on utilise une indentation raisonnable pour faciliter la lecture du programme.

3) Le filtrage

Le filtrage est une autre façon de définir un objet de façon conditionnelle :

```
let f = function
      | 0 -> 1
      | n -> 2*n;;
 val f : int -> int = <fun>
```

ou encore

```
let f n = match n with
      | 0 -> 1
      | _ -> 2*n;;
 val f : int -> int = <fun>
```

— Le filtrage peut contenir autant de cas que souhaité.

— Les cas sont examinés dans l'ordre, le premier qui correspond est appliqué.

— Les cas doivent être du même type.

— Les valeurs renvoyées doivent être du même type.

```
let f cpl = match cpl with
          | (0,0) -> 1
          | (0,y) -> y
          | (x,y) -> x;;
 val f : int * int -> int = <fun>
```

— Il est fortement recommandé que les cas couvrent toutes les possibilités autorisées par ce type de données.

— Si un cas est prévu être impossible lors d'une exécution normale, on peut utiliser la commande `failwith` qui, si elle est exécutée, interrompt l'exécution en affichant un message d'erreur.

— On peut utiliser le symbole `_` pour représenter un objet sans nom ni condition dans un des cas.

```
let f n = match n with
  | 0 -> 1
  | 1 -> 0;;
```

```
      Line 4, characters 10-44:
      4 | ..........match n with
      5 |   | 0 -> 1
      6 |   | 1 -> 0..
      Warning 8: this pattern-matching is not exhaustive.
      Here is an example of a case that is not matched:
      2
      val f : int -> int = <fun>
```

```
let g n = match n with
  | 0 -> 1
  | 1 -> 0
  | _ -> failwith "l'entrée de g est différente de 0 ou 1";;
```

```
     val g : int -> int = <fun>
```

Pour intégrer une condition au filtrage, deux approches sont possibles :
— On peut ajouter une condition booléenne précédée par `when` dans certains des cas.
— On peut faire porter le filtrage sur le résultat d'une fonction.

```
let syracuse1 n = match n with
  | n when n mod 2 = 0 -> n/2
  | n -> 3*n+1;;
```

```
      val syracuse1 : int -> int = <fun>
```

```
let syracuse2 n = match n mod 2 with
  | 0 -> n/2
  | _ -> 3*n+1;;
```

```
      val syracuse2 : int -> int = <fun>
```

La définition d'une fonction par le mot-clé `function` permet d'effectuer directement un filtrage sur l'argument de la fonction (mais cette syntaxe me semble rarement utilisée) :

```
let g = function
  | 0 -> 1
  | 1 -> 0
  | _ -> failwith "l'entrée de g est différente de 0 ou 1";;
    val g : int -> int = <fun>
```

IV Structures itératives

Lorsqu'une opération ou une procédure doit être utilisée un certain nombre de fois on utilise alors des boucles itératives. Deux cas sont à distinguer : lorsqu'on connait le nombre d'itérations à faire ou lorsque le nombre est soumis à une condition.

Pour justifier (correction du programme) il est souvent utile d'utiliser un invariant de boucle. C'est une propriété qui est vraie avant et après chaque itération et qui permet de valider le résultat attendu. Il est souvent justifié par récurrence.

Lorsque l'itération est destinée à faire évoluer des valeurs il est nécessaire d'utiliser des références.

1) itération conditionnelle : boucle `WHILE`

Il s'agit du cas où la répétition est conditionnée par l'évaluation d'une expression booléenne

L'utilisation de telles boucles nécessite une justification claire de la terminaison !

Exemple : `(* cherche le plus petit diviseur > 1 de n, n >= 2 *)`

```
let diviseur n =
    let d = ref 2 in
    while n mod !d <> 0 do d := !d + 1 done;
    !d ;;
```

Exemple : test de primalité

```
let estpremier n =
    let onatrouve = ref false and k = ref 2 in
        while (not !onatrouve)&&(!k * !k<=n) do
          if (n mod !k)=0 then
            onatrouve:=true;
          incr(k);
        done;
        not(!onatrouve);;
```

Exemple : Dans l'algorithme d'Euclide, le pgcd de `!a` et `!b` est conservé, c'est l'invariant de boucle.

```
let pgcd m n =
        let a = ref m and b = ref n in
        while (!a<>0)&&(!b<>0) do
          if !a > !b then a:= !a -!b
            else b:=!b-!a
        done;
        !a+!b;;
```

2) Itération inconditionnelle : boucle `FOR`

Elles sont utilisées lorsqu'on connaît *à l'entrée dans la boucle* le nombre d'itérations à effectuer (en Caml, instructions `for`, le compteur de boucle (ou indice de boucle) est une valeur entière).

Attention : La terminaison d'une boucle `FOR` est assurée par la nature même de la boucle, **à condition** que la valeur du compteur ne soit pas modifiée en cours de boucle .

Exemple : factoriel

```
let fact_iter n = if n=0 then 1 else
    begin
        let s=ref 1 in
        for i=1 to n do s:=!s*i done;
        !s;
    end;;
```

Exemple : affichage des n premiers carrés

```
let carre n =
    for i = 1 to n do
        print_int( i * i);
        print_newline();
    done ;;
```

Exemple : calcul itératif de $\sum_{i=0}^{n} i$

```
let somme n =
    let s = ref 1 in
```

```
        for i = 2 to n do
           s:= !s + i;
        done;
  !s;;
```

V Typage

`Ocaml` possède beaucoup de types différents. Les fonctions de bases qui correspondent se trouvent dans des modules spécifiques. On a la possibilité de ne pas charger le module, dans ce cas il faut le préciser avant la fonction (ou composante) et utiliser la notation pointée `Module.composante`, ou alors le chargé par la directive `open` *module*.

Parmis les types de Caml il y a :

- Le type **unit**, dont l'unique valeur notée () est vide.
- Les booléens **bool** (true et false), n'ayant que les deux valeurs de vérité vrai et faux.
  Les opérateurs sont :
  — `&&` c'est le « et »logique
  — `||` c'est le « ou »logique
  — `not` pour la négation
- Les entiers **int** (les entiers de -2^{30} à $2^{30}-1$)
- Les réels **float** (à ce propos entiers et réels ne sont pas compatibles à moins d'en donner l'ordre par #open "float" dans certaines versions de Caml, sinon + - * / doivent être suivis d'un point).
- Les caractères **char** (encadrés par des apostrophes). Un caractère est codé par un entier compris entre 0 et 255.

```
let a = 'a';;
Char.code a;;
    - : int = 97
```

- Les chaînes de caractères **"string"** (encadrés de guillemets). On peut accéder à un caractère particulier d'une chaîne, mais pas le modifier dans la chaîne :

```
let s = "azerty";;
s.[1] ;;
    - : char = 'z'
```

- Le type **unit** est le type dont l'unique valeur notée () est vide
- Les couples : les deux composantes n'ont pas besoin d'être du même type

```
let cp = (1,'a');;
    cp : int * char = 1, 'a'
fst cp ;;
    - : int = 1
snd cp ;;
    - : char = 'a'
```

- Les n-uplets (pas d'équivalent de `fst` et `snd`, on accède aux différentes composantes par filtrage).
- les **tableaux** (ou appelé aussi vecteurs). Toutes les cases du tableau doivent avoir le même type.

```
let t = [| 1;2;3|] ;;
     val : int array = [|1;2;3|]
```

On peut aussi définir un tableau dont toutes les cases sont égales par sa taille

```
let t = Array.make 10 2;;
    val t : int array = [|2; 2; 2; 2; 2; 2; 2; 2; 2; 2|]
```

On peut construire le tableau à l'aide d'une fonction (il faut faire attention car l'indexation commence à 0) :

```
let f i = i+ 1;;
let t = Array.init 10 f;;
    val t : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

On appelle la i^{eme} composante en temps constant par

```
t.(1) ;;
     - : int = 2
```

La longueur est fixée par la création. Elle ne peut donc pas être changée.
On peut toute fois changer les valeurs par l'affectation suivante :

```
t.(0)<- 2;;
```

- les listes

```
let lst = [1;2;3] ;;
    val lst : int list = [1;2;3]
```

La longueur est dynamique. La recherche d'une composante n'est plus en temps constant car il faut parcourir toute la liste.

On ne peut qu'obtenir deux éléments rapidement, c'est la tête et la queue de la liste :

```
List.hd lst ;;
    - : int = 1
List.tl lst ;;
    - : int list [2;3]
```

Il existe des fonctions de conversion de types : `int_of_float`, `float_of_int`, et de façon générale `type1_of_type2` :

```
string_of_int;;
    - : int -> string = <fun>
```

VI Exercices

Exercice 1 : Prévoir la réponse de l'interprète de commande après les définitions suivantes :

1) .

```
let a = 2 ;;
let f x = a * x ;;
let a = 3 in f 1 ;;
    ???
let a = 3 ;;
f 1 ;;
    ???
```

2) .

```
let a =
        let a = 3 and b = 2 in
            let a = a + b and b = a - b in
                a - b ;;
    ???
let b = 2 in a - b * b ;;
    ???
```

Exercice 2 :

1) En utilisant une affectation locale, calculer

$$\frac{1+\sqrt{7}+\sqrt{7}^3}{1+e^{\sqrt{7}}} \quad , \quad \frac{\ln(\cos(1))+\sin(\ln(5))}{\cos(1)+\ln(5)}$$

2) En utilisant une affectation locale, donner un calcul efficace (avec un seul appel à la fonction exponentielle) de $th(x)$.

Exercice 3 : Ecrire les fonctions suivantes :

1) `funct1` qui à $f \in \mathbb{R}^{\mathbb{R}}$ associe la valeur $\frac{1}{2}(f(0)+f(1))$

2) `funct2` qui à $(f,x) \in \mathbb{R}^{\mathbb{R}} \times \mathbb{R}$ associe la valeur $(f(x))^2$

3) `funct3` qui à $f \in \mathbb{R}^{\mathbb{R}}$ associe la fonction f^2 (le carré de f)

4) `funct4` qui à $f \in \mathbb{R}^{\mathbb{R}}$ associe la fonction $x \mapsto f(x+1)$.

Exercice 4 : L'opérateur des différences finies Δ associe à toute suite $(u_n)_{n\in\mathbb{N}}$ la suite $((u_{n+1}-u_n)_{n\in\mathbb{N}}$. Écrire une fonction `delta` qui réalise cette transformation. On commencera par en donner son type.

Exercice 5 : Un nombre entier $n \geq 2$ est dit parfait s'il est égal à la somme de ses diviseurs positifs (y compris 1) différents de lui-même.

Ecrire une fonction qui donne tous les nombres parfais inférieurs à 10 000.

Exercice 6 : Ecrire une fonction `recherche` d'arguments un élément `e` et un tableau `t` trié, qui recherche présence et la position de l'élément dans le tableau si celui-ci est présent.

Exercice 7 : On suppose qu'un polynôme de degré n, $P = \sum_{k=0}^{n} a_k x^k$ est représenté par un tableau $[|a_0;\ldots;a_n|]$.

Ecrire une fonction `valeur` de signature `int array -> int -> int`, qui calcule $P(a)$. Cette fonction devra être linéaire selon le degré du polynôme.

Exercice 8 : Ecrire une fonction **log2** qui prend en argument un entier n, avec $n \geq 1$, et calcule le plus grand entier k tel que $2^k \leq n$.

On proposera une version itérative et une version récursive.

Exercice 9 : Un nombre d'Armstrong est un nombre qui est égal à la somme des cubes des chiffres de son écriture en base 10 ; par exemple 153 est un nombre d'Armstrong puisque $153 = 1^3 + 5^3 + 3^3$.

Écrire un programme qui calcule tous les nombres d'Armstrong à trois ou quatre chiffres.