LANGAGE OCaml – TP 1
Types – Instructions de base

Introduction

La programmation impérative et la programmation déclarative obéissent à des paradigmes différents. L'approche impérative s'attache à décrire le processus permettant d'aboutir au résultat attendu. L'approche privilégiée en programmation déclarative est la description du résultat attendu plutôt que les étapes permettant l'obtention de ce résultat. Son niveau d'abstraction est donc très élevé, ce qui rend les codes plus synthétiques à l'écriture mais plus difficiles à déchiffrer pour un autre programmeur.

Caractéristique	Impératif	Déclaratif fonctionnel
Vision du programmeur	Comment effectuer les tâches et assurer le suivi des modifications des variables	Informations souhaitées et transformations requises pour obtenir le résultat
Modification des contenus des variables	Important	Inexistant
Ordre d'exécution des instructions	Important	Peu important
Déroulement du processus	Boucles, conditions, appels de fonctions	Appel de fonctions, dont récursivité
Objets manipulés	Structures de données	Fonction comme objet collectant des données

Cependant, si la programmation impérative procède essentiellement par effets de bords, c'est-à-dire par la modification en place des valeurs des variables, les effets de bords et le caractère immuable des structures de données ne sont pas totalement absents de la programmation déclarative. Le langage OCaml utilise en effet les structures usuelles vues en C en début d'année (conditionnelles, boucles (for, while), variables modifiables en place, tableaux, …).

La programmation fonctionnelle est une sous-famille des approches impérative et déclarative. Elle consiste en la composition de problèmes sous la forme d'un ensemble de fonctions à exécuter.

Néanmoins, les frontières entre ces différentes approches sont désormais floues et la tendance actuelle est à la programmation multi-paradigme.

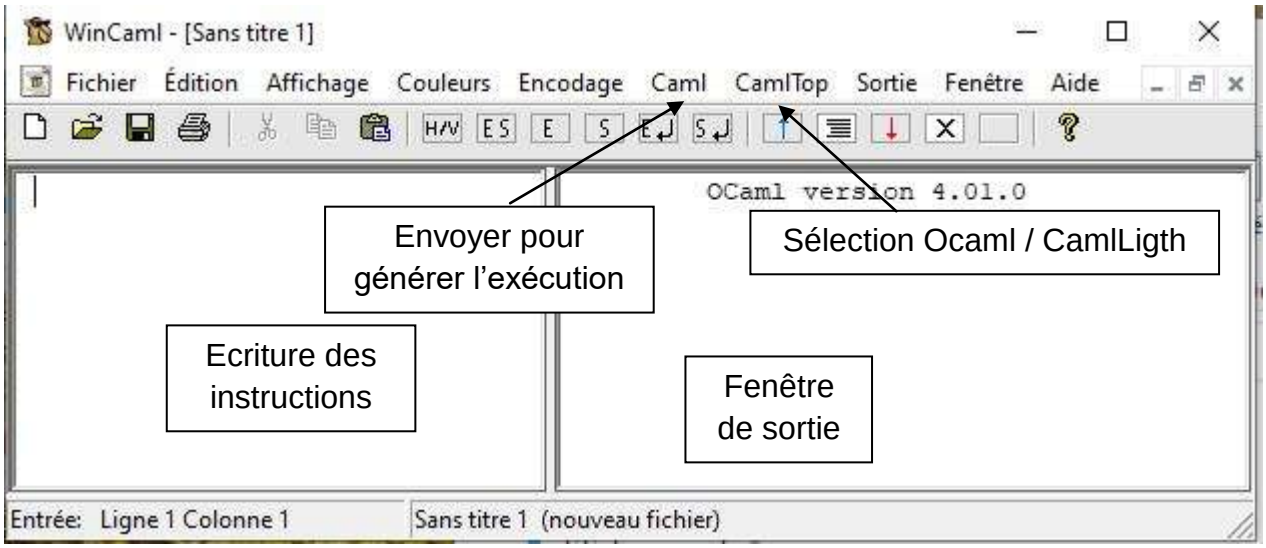
Prise en main de WinCaml

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.

Cliquer sur l'icône permettant de lancer wincaml

Sélectionner fichier, nouveau dans la barre d'outils.

La fenêtre suivante apparaît :



Vérifier que vous êtes bien en Ocaml ou Caml et non en CamlLigth dans CamlTop.

Déclarations de variables

Le langage OCaml est constitué d'une suite d'expressions à évaluer, séparées par un double point-virgule.

Une déclaration affecte le résultat de l'évaluation d'une expression à une variable et est introduite par le mot-clé `let`.

```
let x = 2 + 1 ;;
```

Cette instruction évalue 1 + 2 puis affecte le résultat à la variable `x`.

Remarques

- La variable `x` doit nécessairement être initialisée, c'est-à-dire qu'une valeur doit être liée au nom de la variable ;
- Le type de la variable est inféré par le compilateur ; les entiers et les flottants sont codés suivant la norme IEEE-754 ;
- Le contenu de la variable n'est pas modifiable : `x` contiendra la valeur 3 jusqu'à la fin du programme ; on dit que la variable est immuable ;
- Un type particulier existe en OCaml : le type `unit`. Ce type contient une valeur unique notée `()`, utilisée lorsqu'une instruction ne retourne pas de valeur particulière. Elle correspond au `NULL` ou au `None` de certains langages ;
- Les noms de variables doivent toujours commencer par une minuscule.

Exemple : Taper les instructions suivantes, les exécuter ligne par ligne (sélectionner une ligne en positionnant le curseur dessus puis envoyer) et analyser leur effet.

```
let a = 2 + 1 ;;
```

```
print_string "La valeur de x est : \n" ;;
print_int a ;;
print_newline();;
a ;;
(* l'instruction print, permettant d'afficher n'importe quelle
variable n'existe pas ;
pour afficher un entier : print_int,
pour afficher une chaîne de caractères, print_string
pour afficher un float : print_float
Les parenthèses sont inutiles : print_int (x) ;; mais ne génèrent pas
d'erreur
Et donc, les commentaires se mettent entre les symboles (* et *) *)

a = a + 2 ;;
print_int a ;;

let a = 5 ;;
let b = a * a ;;
let c = a + b ;;

let w = 3.0 and x = 2. ;;
(* Type inféré : float *)
(* le and permet l'affectation multiple *)
let y = w + x ;;
let z = w +. x ;;
(* d'où l'importance des espaces de part et d'autre de l'opérateur,
pour plus de lisibilité *)
let u = 2. and v = u + 5 ;;
(* v ne peut pas utiliser u pour son initialisation car u est en cours
de création *)
let z = 3 *. y ;;
```

Les opérateurs mathématiques usuels ne sont pas surchargés : ils ne peuvent s'appliquer qu'entre de deux variables d'un type particulier.

Si l'on souhaite définir une variable modifiable en place, il faut introduire une référence (équivalent à un pointeur en C). Comme les variables non modifiables, la référence doit être initialisée et son type est inféré par le compilateur. Elle introduite par le mot-clé `ref` et l'accès à son contenu se fait en utilisant `!x` (`*x` en C ; distinction entre le pointeur et son contenu). Même si cette possibilité existe, il convient de l'utiliser avec parcimonie : les concepts sous-jacents au langage Caml consistent en des variables non mutables et des évaluations de fonctions.

Exemple : Taper les instructions suivantes, les exécuter ligne par ligne et analyser leur effet.

```
let x = ref 1 ;;
(* Pas d'opérations après le ref *)
print_int x ;;
print_int !x ;;
x := !x + 2 ;;
(* Attention au := pour changer la valeur contenue dans x *)
print_int !x ;;
x:= 1 ; 5 + !x ;; (* le ; simple indique que l'expression n'est pas
achevée *)
```

Opérateurs mathématiques

Certains opérateurs ne sont pas surchargés : l'addition, la soustraction, la multiplication et la division ne sont pas codées pour les opérations entre entiers ou les opérations entre flottants. Ces opérations entre types différents ne sont donc pas possibles.

	Int	**Float**
Opérations de base	`+, -, *, /`	`+., -., *., /.`
Quotient et reste de division euclidienne	`/, mod`	
Trigonométrie		`cos, sin, tan`
Logarithmes		`log` (équivalent à ln) `exp` (exponentielle) `**` (puissance) `sqrt` (racine carrée)

Des fonctions existent pour forcer le typage d'une variable : int_of_float et float_of_int. Leur utilisation doit rester exceptionnelle.

Opérateurs de comparaison et booléens

Les opérateurs de comparaison sont `<>`, `=`, `<`, `>`, `<=` et `>=` . La syntaxe est identique pour les entiers et les flottants mais ces opérateurs ne peuvent effectuer de comparaisons qu'entre variables du même type.

Le type renvoyé par ces opérateurs de comparaison est `bool` (booléen). Ces variables peuvent prendre deux valeurs, `true` ou `false`.

Opérateurs logiques

Conjonction (et logique)	&&
Disjonction (ou logique)	||
Négation	`not`

Les opérateurs logiques renvoient des booléens.

Comme en C, l'évaluation des équations logiques est une évaluation paresseuse.

Structure conditionnelle

La syntaxe d'une structure conditionnelle est :

```
if condition then expression1 else expression2 ;;
```

Dans le cas où plusieurs instructions doivent être effectuées, on utilise les syntaxes suivantes :

```
if condition then (
      expression1 ;
      expression2 ;
      …)
else (
      expression3 ;
      expression4 ;
      …) ;;
```

```
if condition then begin
      expression1 ;
      expression2 ;
      …
end else begin
      expression3 ;
      expression4 ;
      …
end ;;
```

Remarque

La branche `else` d'une conditionnelle renvoie un type unit de valeur () lorsque le `else` est absent. Ainsi, l'expression suivante est incorrecte du point de vue du typage en OCaml :

```
2 + ( if condition  then 1) ;;
```

Car le compilateur affectera un type `unit` à l'évaluation du second opérande si la condition est fausse. L'addition est alors impossible.

Expressions et instructions

En programmation déclarative fonctionnelle, il n'y a pas de distinction entre expressions et instructions, contrairement à ce qui se fait en programmation impérative, pour laquelle ces deux catégories sont distinctes (une conditionnelle `if-else` ou une boucle `for` ne peuvent être acceptées en position d'expression, et inversement).
Certaines constructions telles que les déclarations ou les appels de fonctions peuvent néanmoins apparaître en temps qu'instruction ou expression dans le paradigme impératif.

En OCaml, les constructions utilisées en programmation impératives sont considérées comme des expressions à évaluer. On peut donc écrire :

```
1 + (if condition then expression1 else expression2) ;;
```

Le premier opérande est évalué et, si la condition est true, le second opérande est évalué comme étant le résultat de *expression1*, sinon, *expression2* est évalué et le résultat est affecté au second opérande. Les deux expressions doivent donc impérativement être du même type.

Variables locales

En C, la portée d'une variable est limitée au bloc au début duquel elle est déclarée. Ce bloc est délimité par des accolades.

En OCaml, la notion de variable locale n'est pas associée à une notion de bloc. Elle est introduite par la construction `let in` qui introduit une variable localement, dans une expression :

```
let x = 10 in 2 * x ;;
```

Comme pour les variables globales, son type est inféré et sa valeur immuable. Sa portée se limite à l'expression suivant le `in`.

La construction `let in` est une expression comme une autre. On peut ainsi écrire :

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

Cette variable locale peut être définie comme une référence de manière à être modifiable en place :

```
let x = ref 1 in
x := !x + 1 ;          (* un seul ; : expression non terminée *)
let y = !x * !x in
print_int y ;
print_newline() ;
print_int x ;;         (* fin de l'expression marquée par ;; *)
```

Fonctions

Déclaration d'une fonction

La syntaxe d'une déclaration d'une fonction est la suivante :

`let carre x = x**2. ;;`	Définit une fonction nommée `carre`, prenant un unique argument `x` et renvoyant le carré de `x`. Le corps de la fonction correspond à l'expression qui doit être évaluée lors de l'appel de la fonction. Aucun return n'apparaît puisqu'il n'y a pas de distinction entre expression et instruction en OCaml. Les arguments ne sont pas mis entre parenthèses et les types sont inférés.

L'envoi de cette fonction provoque la sortie : `val carre : float -> float = <fun>`

Le système indique que l'on a déclaré une variable nommée `carre`, que son type est `float -> float`, c'est-à-dire le type d'une fonction prenant en argument une variable de type `float` et renvoyant une valeur de type `float` et que sa valeur est `<fun>`, c'est-à-dire une fonction.

Pour appeler cette fonction et lui passer un argument, on tape l'instruction : `carre 4. ;;`

La sortie est alors : `- : float = 16.`

Analyser le comportement de la fonction carre pour les appels suivants :

```
carre 1. +. 1.;;
carre (1. +. 1.);;
```

Fonction à plusieurs arguments

Une fonction peut aussi prendre plusieurs arguments ; ils sont alors mentionnés sans parenthèses :

`let moyenne x y = (x +. y) /. 2.0 ;;`	Le type est inféré : cette fonction prend deux arguments de type `float` et renvoie une valeur de type `float`. C'est la présence du `+.` qui permet au compilateur d'inférer le type.
`val moyenne : float -> float -> float = <fun>`	Résultat de l'envoi de la déclaration ci-dessus.
`moyenne 4 5.2 ;;`	Renvoie une erreur car les deux arguments passés ne sont pas de type `float`.
`moyenne (4. 5.2) ;;`	Renvoie un message d'erreur : les parenthèses servent à définir un type particulier (n-uplet). La fonction ne reçoit pas en argument deux variables de type `float`.
`moyenne 4. 5.2 ;;`	Affichage en sortie : `- : float = 4.6`
`moyenne (4. +. 2.) 2. ;;`	Affichage en sortie : `- : float = 4.`

`moyenne 4.+.2. 2. ;;`	Affichage en sortie :
	`moyenne 4.+.2. 2.;;`
	`  ^^^^^^^^^^`
	`Error: This expression has type float -> float`
	`  but an expression was expected of type float`

Il existe une subtilité liée aux notations.

`let h (x, y) = x + y ;;` `h (1,2);;`	Signature de h : Sortie :
`let k x y = x + y;;` `k 1 2;;`	Signature de k : Sortie :

Dans le cas de la fonction h, la fonction prend en argument un couple de deux entiers. La fonction a pour signature : `int * int -> int = <fun>` ; le symbole * indique qu'il s'agit d'un couple de valeur. IL s'agit donc d'une fonction d'une variable (donc de N dans N).

Dans le cas de la fonction k, la fonction prend en argument deux entiers. Sa signature est `int -> int -> int = <fun>` ; il s'agit d'une fonction de plusieurs variables (donc de N^2 dans N).

Exercice 1

Prévoir l'affichage produit par les instructions suivantes. Vérifier *ensuite* vos prédictions en écrivant et exécutant ces lignes dans OCaml.

```
let a = ref 9;;
a := 1 ; 2+ !a ;;
3* (a := 7 ; 2 + !a);;
```

Exercice 2

Evaluez sur papier le résultat des expressions suivantes. Dans le cas où une erreur est détectée, proposez une correction.
Valider ensuite vos résultats.

```
let x = 2 in x + 2 ;;
let x = 1 and y = 3 in let x = 2 in x + y ;;
let x, y, z = 1, 2, 3 in x + y + z ;;
let x, y, z = true, 2, 3 in x || y = z ;;
let x, y, z = true, 2, 3 in x and y < z ;;
```

Exercice 3

Une année bissextile est une année comportant 366 jours au lieu de 365 jours pour une année commune. Le jour supplémentaire, le 29 février, est placé après le dernier jour de ce mois qui compte habituellement 28 jours dans le calendrier grégorien, apparu en 1582.
Ce genre d'année existe pour compenser la différence de temps entre l'année calendaire (365 jours) et l'année solaire, c'est-à-dire le temps pris par la Terre pour effectuer une révolution complète autour du Soleil, qui est 365,242 jours. Un jour surnuméraire est donc ajouté régulièrement pour que la moyenne de la durée des années calendaires soit la plus proche possible de l'année solaire. Sans cette correction, la date des saisons se décalerait progressivement dans le calendrier.

Depuis l'ajustement du calendrier grégorien en 1582, l'année n'est bissextile que dans l'un des deux cas suivants :

- si l'année est divisible par 4 et non divisible par 100 ;
- si l'année est divisible par 400 (« divisible » signifie que la division donne un nombre entier, sans reste).

Sinon, l'année n'est pas bissextile : elle a la durée habituelle de 365 jours (elle est dite année commune).
Ecrire une suite d'instruction permettant de tester si une année est bissextile ou non. Prévoir l'affichage du résultat.

Exercice 4

Dans cet exercice, on prendra $\pi = 3{,}14$.

Ecrire et tester une fonction de signature `perimetre : float -> float = <fun>` prenant en argument le rayon d'un cercle et renvoyant son périmètre ($2 \times \pi \times r$).

Ecrire et tester une fonction de signature `surface : float -> float = <fun>` prenant en argument le rayon d'un cercle et renvoyant sa surface ($\pi \times r^2$).

Ecrire et tester une fonction de signature `volume : float -> float -> float = <fun>` prenant en argument le rayon et la hauteur d'un cylindre et renvoyant sa surface extérieure. Cette fonction devra impérativement utiliser la fonction `surface`.

Ecrire et tester deux fonctions de signature `surface_ext_0 : float -> float -> float = <fun>` et `surface_ext : float -> float -> float = <fun>` prenant en argument le rayon et la hauteur d'un cylindre et renvoyant sa surface extérieure. Ces fonctions devront impérativement utiliser les fonctions `perimetre` et `surface`. La première fonction ne définira aucune variable locale ; la seconde utilisera la surface de base et le périmètre comme variables locales.