**Intro à la programmation en Caml**

I Introduction

1. Spécificité Caml

1.1 Paradigme

Caml est un langage essentiellement fonctionnel (qui permet aussi de faire de la programmation impérative).

La programmation impérative indique au processeur les étapes à suivre.

La programmation fonctionnelle met en avant la définition et l’évaluation de fonctions et limite les affectations.

Les fonctions sont considérées comme des valeurs, elles peuvent être le paramètre ou le résultat d’autres fonctions.

Ex : Calcul de factoriel n >= 1

- style impératif

fact(n) :

res = 1 # résultat

pour i = 1 … n

res \*= i

return res

- style fonctionnel:

fact(n):

fact(1) = 1

fact(n) = n \* fact(n-1)

1.2 Typage

Caml est fortement type car tous les objets manipulés appartiennent à un type donné qui ne change pas. On dit que les conversions de type sont interdites.

Au contraire, Python est faiblement typé.

Ex :

i = (i + j) / 2 # i peut devenir float

En Caml impossible, l’entier 1 s’écrit 1 et le réel 1 s’écrit 1.

Le typage d’une fonction en Caml est réalisé lors de sa définition.

Caml détermine le type des arguments et du résultat.

Ex :

Addition des entiers +

Réels +.

1 + 2. (bug)

TODO : (float\_of\_int 1) +. 2. (à éviter)

1.5 + 2.5 (erreur)

TODO: 1.5 +. 2.5

sin 2.

Work: 3/5 (result 0 – division entière) (3 /. 5 doesn’t work) 6 / 5 => 1

(Ben: 3. /. 5. => 2.5)

2. Variables globales:

Règles pour le nom des variables :

- commence par une lettre

- peut utiliser des chiffres

- pas d’espace

- tiret bas uniquement

2.1. Variables globales :

On attribue un nom à une variable avec l’instruction let

let pi = 3.14159;;

pi \*. 2.;;

Une fois crée la valeur ne change plus sauf en la modifiant directement.

let a = 1;; let b = a + 1;;

let a = 2;;

2.2. Variables locales:

let a = 1 in 2 \* a;;

Si on appelle ça on a une erreur.

(a est inconnue)

let a = 1;;

let a = 2 in 2\*a;;

a;;; (vaut 1)

2.3 Définitions multiples:

let a = 1 and b = 2 in a \* b;;

Attention on ne peut pas faire de définitions multiples liées.

On ne peut pas faire : let a = 2 and b = 2 \* a in a \* b;; (erreur a inconnue)

let a = 2 in let b = 2 \* a in a \*b;;

3 Types de base:

3.1 Types simples:

1. int (entiers compris entre -2^30 + 1 et 2^30 – 1)

Opérateurs + \* / -

Modulo et valeur absolue : mod abs

abs (-4);; (use parenthesis when -)

5 mod 2;;

2. float (nombres décimaux approchés)

Opérateurs exp log (ln) sqrt sin cos tan +. -. /. \*. (B : tanh)

(int\_of\_float … éviter – reprendre en amont le problème si doit le faire en général)

3. Bool (valeurs booléennes true false)

let a = 1 = 2;; // a:bool = false (un égal pour la comparaison)

opérateurs && || not (pas and, or fonctionne mais pas and donc éviter or)

Evaluation paresseuse : le second test n’est fait que si nécessaire

let a = false && (1/0 = 1);; (dis pas erreur (ne fait pas la division par 0))

4. char (caractère let a = ‘z’ ;;)

char\_of\_int int\_of\_char (conversion code ascii)

5. string (chaîne de caractères)

let s = “mpsi”;;

s.[0];; // char ‘m’

s.[0] <- ‘M’;; function de type unit

s;; string “Mpsi”

6. unit () (note comme ça)

3.2 Couples, triplets, p-uplet:

On crée les p-uplets en séparant avec les objets avec des virgules

let a = (true, 1, 2.35, « bonjour » );;

a : bool \* int \* float \* string

4. Fonctions

4.1 Fonctions d’une seule variable

let f = function x -> x \* x;;

let f = fun x -> x \* x;; // sometimes

let f x = x \* x;; // prefer this si \*. float -> float

f est de type int (entrée) -> int (sortie)

let f x = x \* sin x;; // error, patch: \*.

Les parenthèses sont parfois indispensables.

let f x = x + 1;; // f: int -> int

f 2 \* 3;; compris comme (f 2) \* 3 (priorité fonctions) mieux un peu surparenthèser

-> 9

Autrement dit f x y est interprété en (f x) y

4.2 Fonctions

Deux solutions pour définir une fonction à plusieurs variables. Une première possibilité consiste à définir une fonction prenant en argument un p-uplet

Ex :

On veut écrire une fonction divise qui dit si m divise n.

let divise (m, n) = n mod m = 0;;

divise: (int \* int) -> bool

Le type est détermine automatiquement quand c’est possible. Parfois le type n’est pas connu à l’avance, on parle de fonctions polymorphes.

let fst (a, b) = a;;// snd (second) f : (‘a \* ’b) -> ‘a

On pouvait changer le b par \_ si on ne s’en sert pas

Le type des arguments peut aussi être une fonction.

Ex :

Fonction acc qui calcule le taux d’accroissement d’une fonction entre a et b ?

let acc (f, a, b) =

(f(b) -. f(a)) /. (b -. a);; // (float -> float, float, float) -> float

Si on a:

let acc2 (f, a, b) = f(b) -. f(a);; // (‘a -> float \* ‘a \* ‘a) -> float // (‘a: alpha)

De même on peut renvoyer une fonction :

let double f = function x -> 2 \* (f x) ;; // *(‘a -> int)* -> *‘a -> int*

argument résultat

let g = double somme;;

4.3 Curryfication et décurryfication

Ex :

(x, y) appartenant à Z et x + y aussi

On veut définir f :x,y -> x + y

~~let somme (x, y) = x + y;; // (int \* int) -> int (moyen)~~

let somme x y = x + y;; // int -> int -> int

La première version est une fonction a une seule variable qui est un couple.

La seconde version est la version curryfiée et c’est celle qu’on préfère car elle permet de définir des applications partielles. (B : ?)

D’un point de vue mathématique, on utilise (l’existence d’une bijection entre les deux ensembles) :

( Z x Z -> Z ) et ( Z -> F(Z, Z) )

( (x, y) -> x + y ) ( x -> Z -> Z )

y |-> x+y)

let somme2 = somme 2;; // int -> int

Syntaxe équivalente :

let mul = function m -> function n -> m \* n;; (nul)

let mul m = function n -> m \* n;; (nul)

let mul m n = m \* n;;

int –> int -> int

/!\ mul (3, 2) -> erreur

Attention à l’ordre des arguments.

let ascii c n = int\_of\_char c = n;; char -> int -> bool (= compare des choses de même type)

ascii ‘a’ 97 (true ?)

let g = ascii 97;; // erreur

let g = function c -> ascii c 97;;

ou pref: let g c = ascii c 97;;