Option informatique (cours de $1^{ère}$ année)

Éric Le Nagard

9 mai 2003

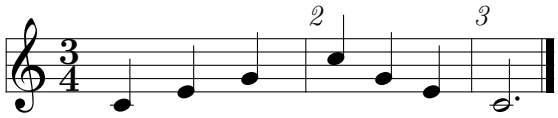
1 Introduction

Le but du cours le l'option d'informatique est une présentation et une mise en œuvre d'algorithmes et de structures de données pour résoudre certains problèmes. Un algorithme, c'est suivant le dictionnaire hachette une «Méthode de résolution d'un problème utilisant un nombre fini d'applications d'une règle».

Un algorithme permet donc à partir d'une condition initiale d'appliquer différentes opérations dans un ordre déterminé et en nombre fini pour arriver à un résultat. Comme exemple d'algorithme on peut citer les recettes de cuisine qui, dans le principe, permettent d'obtenir en un nombre fini d'étapes des mets succulents. On peut aussi citer le cas des partitions de musique où chaque séquence de l'algorithme correspond à l'exécution d'une ou plusieurs notes pendant un temps déterminé avec la bonne nuance. Une notice de montage fournit aussi un exemple d'algorithme pour les amateurs de Lego ou de Meccano.

Dans le cadre du cours de l'option informatique, nous allons décrire des algorithmes, les mettre en œuvre, mais aussi réfléchir à leur efficacité, c'est à dire évaluer la façon dont va évoluer le temps d'exécution de l'algorithme en fonction des données.

Pour décrire tout algorithme, il faut disposer d'un langage qui sera compréhensible par celui qui va l'exécuter. Et tout langage demande un apprentissage. Si vous ne savez ce que veut dire «blanchir les zestes de deux oranges» vous ne réussirez pas la recette du canard à l'orange, de même si vous ne savez pas ce que signifie :



vous aurez bien du mal à nous jouer ce petit air de musique écrit par un compositeur anonyme.

Dans le cadre de l'option informatique, nous décrirons les algorithmes en français mais aussi dans un langage compris par l'ordinateur : Caml.

2 Caml

Pour vous parler de Caml, voici un extrait de la FAQ (Foire aux questions) qui donne les questions et les réponses les plus fréquemment posées sur Caml.

2.1 Qu'est-ce que Caml ?

Caml est un langage de programmation. C'est un langage fonctionnel car les briques de base des programmes sont les fonctions. C'est un langage fortement typé, ce qui signifie que les objets manipulés appartiennent à un ensemble identifié par un nom qu'on appelle son type. En

Caml, les types sont gérés et manipulés par la machine, sans intervention de l'utilisateur (types synthétisés).

Le langage est disponible sur à peu près toutes les machines Unix, sur PC (à partir de 386) et sur Macintosh. Une utilisation confortable nécessite 2 MO de mémoire libre et 2 MO de mémoire sur disque dur.

2.2 Que signifie le nom "Caml" ?

"Caml" est un acronyme en langue anglaise: "Categorical Abstract Machine Language", c'est-à-dire langage de la machine abstraite catégorique. La CAM est une machine abstraite capable de définir et d'exécuter les fonctions, et qui est issue de considérations théoriques sur les relations entre la théorie des catégories et le lambda-calcul. Le premier compilateur du langage générait du code pour cette machine abstraite (en 1984).

La deuxième filiation de ce nom est ML (acronyme pour Meta Language): Caml est aussi issu de ce langage de programmation crée par Robin Milner en 1978, et qui servait à programmer les tactiques de preuves dans le système de preuves LCF.

2.3 Est-ce un langage compilé ou interprété ?

Caml est un langage compilé. Cependant tous les systèmes Caml (on entend par "système" l'ensemble compilateur+bibliothèques associées) 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.

2.4 Où trouver de la documentation ?

Livres : 3 livres d'introduction en Français.

* Pour le programmeur (introduction progressive au langage, nombreux exemples de taille conséquente, orientation programmation): Pierre Weis et Xavier Leroy, Le langage Caml (Intereditions, ISBN 2-7296-0493-6).

* Pour le professeur et le programmeur intéressé par la théorie des langages: Thérèse Hardin et Véronique Donzeau-Gouge, Concepts et Outils de Programmation – le style fonctionnel, le style impératif avec Caml et Ada (Intereditions, ISBN 2-7296-0419-7).

* Enfin pour une orientation algorithmique exprimée dans un langage fonctionnel: Guy Cousineau et Michel Mauny, approche fonctionnelle de la programmation (Ediscience, ISBN 2-84074-114-8).

* Un manuel de référence du langage: Xavier Leroy et Pierre Weis, Manuel de Référence du langage Caml (Intereditions, ISBN 2-7296-0492-8).

* consultez le serveur de l'INRIA au sujet de Caml:

  http://pauillac.inria.fr/caml/index-fra.html

3 environnement de travail

présentation sur ordinateur

```
#1+1;;
```

```
- : int = 2
```

Le système reconnaît que le résultat est un entier. Caml est un langage dit fortement typé. cela présente un certain nombre d'avantages comme la reconnaissance automatique de type mais aussi un certain nombre d'inconvénients comme celui-ci :

```
#1+2.1;;
Entrée interactive:
>1+2.1;;
>  ^^^
Cette expression est de type float,
mais est utilisée avec le type int.
```

Pour effectuer le calcul précédent, il faut l'écrire de la manière suivante :

```
#1. +. 2.1;;
- : float = 3.1
```

L'addition sur les entiers n'est donc pas la même que l'addition sur les réels.
Pour créer, une constante Calm, la syntaxe est du type : `let` *nom* `=` *valeur* `;;`

```
#let n = 10;;
n : int = 10
```

4 Les types de bases

4.1 les booléens

les booléens de base sont `true` pour vrai et `false` pour faux,
opérations sur les booléens : `or` ou `||` (pour le «ou»), `&` ou `&&` (pour le «et»), `not` (pour le «non»).
exemples :

```
# true & false;;
- : bool = false
# true or false;;
- : bool = true
# not true;;
- : bool = false
```

4.2 les entiers

Attention : ce sont pas les entiers de Maple, les calculs sont faits modulo 2^{31} ou 2^{63} suivant les machines.
Ainsi nous avons le résultat surprenant suivant :

```
#let n = 1073741823;;
n : int = 1073741823
#n+1;;
- : int = -1073741824
#2*n;;
- : int = -2
```

les opérations sur les entiers :

l'addition : + (exemple : $2 + 3$)

la soustraction : $-$ (exemple : $2 - 3$)

la multiplication : $*$ (exemple : $2 * 3$)

la division euclidienne : / pour obtenir le quotient et `mod` pour obtenir le reste

exemple : 16/3 donne le quotient de la division euclidienne de 16 par 3 et `16 mod 3` donne le reste de la division euclidienne de 16 par 3.

ainsi, pour tout couple d'entiers (x, y) : $x = (x/y) * y + (x \bmod y)$

les fonctions de comparaison sur les entiers :

<, > : pour les inégalités strictes

>=, <= : pour les inégalités larges

<> pour $\neq$

chacune de ces fonctions de comparaison retourne un booléen

exemple :

```
#2 > 3;;
- : bool = false
```

4.3 les réels

Attention : ce sont pas les réels de Maple, ils sont comparables aux réels des calculatrices.

les opérations sur les réels :

l'addition : +. (exemple : 2.5 +. 3.8)

la multiplication : $*$. (exemple : 2.5 *. 3.8)

la soustraction : $-$. (exemple : 2.5 -. 3.8)

la division : /. (exemple : 2.5 /. 3.8)

les fonctions trigonométriques de bases :

cos et *sin* pour les fonctions cosinus et sinus

exp et *log* pour les fonctions exponentiel et logarithme

4.4 conversions réels/entiers

Deux fonctions de conversions réels/entiers `float_of_int` et `int_of_float`.

```
#float_of_int 23;;
- : float = 23.0
#int_of_float 2.5;;
- : int = 2
#int_of_float (-. 2.5);;
- : int = -2
```

4.5 les chaînes de caractères

Les chaînes de caractères sont de la forme "$c_1c_2 \dots c_n$" où les c_i sont des caractères. la chaîne de caractère vide est "". L'opération de concaténation de deux chaînes est obtenue avec l'opérateur binaire $\wedge$.

```
#let mot1 = "toto ";;
mot1 : string = "toto "
#let mot2 = "est  content";;
mot2 : string = "est  content"
```

```
#mot1 ^ mot2;;
- : string = "toto est  content"
```

On dispose aussi de fonctions de conversion d'entiers ou de réels sous forme de chaînes de caractères ainsi que des opérations inverses : `string_of_int`, `string_of_float`, `int_of_string`, `float_of_string`.

```
#int_of_string "12245";;
- : int = 12245
#string_of_int 12245;;
- : string = "12245"
```

On peut obtenir la longueur d'une chaîne de caractères à l'aide de la fonction `string_length` et obtenir ou changer la $i^{ième}$ lettre[1] d'un mot comme le montre l'exemple ci-dessous :

```
#mot1.[1];;
- : char = 'o'
#mot1.[1] <- 'u';;
- : unit = ()
#mot1;;
- : string = "tuto "
#string_length mot1;;
- : int = 5
```

4.6 les tableaux

Les tableaux Caml sont de la forme $[|e_0; e_1; \ldots; e_{n-1}|]$ où les e_i sont tous nécessairement du même type. n est la longueur du tableau, les éléments du tableau étant indexés de 0 à $(n-1)$. $[||]$ constituant le tableau vide de longueur 0.

```
#let t = [|1;2;3;4;6;7|];;
t : int vect = [|1; 2; 3; 4; 6; 7|]
#t.(2);;
- : int = 3
#t.(2) <- 8; t;;
- : int vect = [|1; 2; 8; 4; 6; 7|]
#let tt = [|1;5.5;9;2;"toto"|];;
Entrée interactive:
>let tt = [|1;5.5;9;2;"toto"|];;
>             ^^^
Cette expression est de type float,
mais est utilisée avec le type int.
```

4.7 le type «unit»

Le type «`unit`» correspond à l'ensemble mathématique . On rencontre fréquemment des fonctions qui ne retourne rien comme par exemple les fonctions d'affichage.

1. La première lettre d'un mot est numérotée 0 et la dernière $(n-1)$ où n est la longueur du mot.

4.8 les fonctions d'affichage

Voici les différentes fonctions d'affichage à l'écran :

- `print_newline ()`: qui permet de passer à la ligne,
- `print_int` : pour afficher un entier,
- `print_float` : pour afficher un réel,
- `print_string` : pour afficher une chaîne de caractères,
- `print_char` : pour afficher un caractère

5 éléments de programmation

5.1 `if then else`

La syntaxe est la suivante : `if` *cond* `then` *bloc d'instruction 1* `else` *bloc d'instruction 2*; Les deux blocs d'instructions doivent nécessairement avoir des résultats de même type. La partie «`else` *bloc d'instruction 2*» est facultative mais dans ce cas le résultat du bloc d'instructions du «`thent`» doit être de type `unit`.

```
#if 2 > 3 then 45 else 86;;
- : int = 86
#if 2 > 3 then 45;;
Entrée interactive:
>if 2 > 3 then 45;;
>              ^^
Cette expression est de type int,
mais est utilisée avec le type unit.
```

5.2 boucle inconditionnelle : `for i = 1 to n do ... done;`

La syntaxe d'une boucle inconditionnelle est la suivante :

`for` *nom de variable* `=` *valeur initiale* `to` *valeur finale* `do` *bloc d'instructions* `done`;

La variable de boucle est une variable locale dont le champ d'application est limité au bloc d'instructions de la boucle. On ne peut pas changer la valeur d'une variable de boucle. Celle-ci est automatiquement incrémentée en sortie de boucle. Le pas d'incrément est nécessairement égal à 1. Si la valeur finale est inférieure stricte à la valeur initiale alors le bloc d'instructions n'est pas exécuté. Les valeurs finale et initiale sont nécessairement de type entier.

```
#for i = 1 to 10 do print_int i done;;
12345678910- : unit = ()
#for i = 1 to sqrt(12.) do print_int i done;;
Entrée interactive:
>for i = 1 to sqrt(12.) do print_int i done;;
>             ^^^^^^^^
Cette expression est de type float,
mais est utilisée avec le type int.
```

Il existe une variante de la boucle inconditionnelle avec un pas de -1. dont la syntaxe est la suivante :

`for` *nom de variable* `=` *valeur initiale* `downto` *valeur finale* `do` *bloc d'instructions* `done`;

Dans ce cas, il faut que la valeur initiale soit supérieure à la valeur finale si on veut que le bloc d'instructions soit exécuté au moins une fois.

```
#for i = 21 downto 10 do print_int i done;;
212019181716151413121110- : unit = ()
```

5.3 boucle conditionnelle : `while` *test* `do ... done`

La syntaxe de la boucle conditionnelle est la suivante :

`while` *test booléen* `do` *bloc d'instructions* `done`

Le bloc d'instructions est exécuté tant que le test booléen est vrai.

6 Les constantes et les variables

6.1 les constantes

Nous avons déja vu que pour créer une constante Calm, la syntaxe est la suivante : `let` *nom* `=` *valeur* `;;` On peut définir des constantes qui ont une valeur locale sous la forme : `let` *nom* `=` *valeur* `in` *bloc d'instructions*

6.2 les variables

Pour créer une variable, la syntaxe est la suivante : `let` *nom* `=` `ref` *valeur* `;;` Par exemple,

```
#let a = ref 0;;
a : int ref = ref 0
```

Dans ce cas, a représente la variable et donc le contenant tandis que $!a$ représente le contenu de a.

```
#a;;
- : int ref = ref 0
#!a;;
- : int = 0
```

Pour modifier le contenu d'une variable, la syntaxe est la suivante :

nom `:=` *valeur* `;`

Attention : on ne peut pas changer le type du contenu d'une variable.

```
#a:=5;;
- : unit = ()
#5;;
- : int = 5
#a:=5.5;;
Entrée interactive:
>a:=5.5;;
>   ^^^
Cette expression est de type float,
mais est utilisée avec le type int.
```

7 les fonctions

Les syntaxes de définitions des fonctions sont proches de celles des mathématiques :

$$\boxed{\text{Soit } f : x \mapsto x+1,}$$

s'écrit en `Caml` :

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

On remarque que le résultat est nommé (`f`), typé (`int -> int`), et a une valeur interne non précisée (`<fun>`) [2]

La structure d'une telle définition est :

`let` *nom* `=` `function` *variable* `->` *résultat* `;;`

Il existe cependant en mathématiques d'autres manières de définir une fonction :

$$\boxed{\text{Soit } f(x) = x+1,}$$

qui peut s'écrire en `Caml` :

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

L'objet construit est alors automatiquement une fonction. On peut aussi éviter les parenthèses, avec la forme :

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

Attention : en ce cas, l'appel de la fonction ne se fera pas de la même manière que précédemment, il faudra taper `f 3`, au lieu de `f(3)`.

Ce qui est intéressant dans `Caml`, c'est que les fonctions sont des *êtres* comme les autres.

```
#let composition f g = function x -> f(g(x));;
composition : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#composition f f;;
- : int -> int = <fun>
```

Ce dernier exemple appelle plusieurs remarques :

Remarques :

1. On manipule des êtres abstraits pouvant avoir des types diverses, notons cependant que `Caml` reconnaît que le type d'arrivée de g (`'a`) *doit* être le même que celui de départ de f.
2. Le f qui est dans la *définition* de `composition` est une lettre sans signification (paramètre local) qui n'a aucun rapport avec le f que nous avons défini à plusieurs reprises, et dont seulement la dernière valeur reste connue... En revanche, dans l'*appel* de `composition`, c'est bien notre bien-aimé f qui est utilisé.

2. On n'a pas accès aux codes des fonctions.

3. Le résultat est présenté sous forme d'une fonction donnant une fonction donnant une fonction, essayons de voir sur des exemples plus simples le pourquoi et l'intérêt.

```
#let somme1 (x,y) = x+y;;
somme1 : int * int -> int = <fun>
#let somme2 x y = x+y;;
somme2 : int -> int -> int = <fun>
#somme1 3;;
Entrée interactive:
>somme1 3;;
>       ^
Cette expression est de type int,
mais est utilisée avec le type int * int.
#somme2 3;;
- : int -> int = <fun>
```

Montre bien la différence entre les fonctions à paramètres parenthèsées, qui agit sur un type *composé* (`int * int`), alors que la version non parenthèsée est considérée comme la cascade :

$$x \mapsto (y \mapsto f(x, y)).$$

Ce qui peut se constater avec la définition de la fonction partielle `somme2 x`, où il est bien sûr indispensable que x ait une valeur...