

NOTIONS DE BASE

Mardi 25 Septembre 2012

Option Informatique
Ecole Alsacienne

BONJOUR

- De nouveaux venus ?
- Mise à jour du site Internet
- Avez-vous vos identifiants pour vous connecter ?
- D'autres questions ?

PROGRAMME DE LA SÉANCE

1. Algorithmes
2. OCaml et pseudo-code
3. Variables et typage
4. Constructions classiques
5. Fonctions
6. Arguments

ALGORITHMES

UNE RECETTE DE CUISINE



Leçon du jour : le crumble aux pommes

UNE RECETTE DE CUISINE

1. Eplucher les 6 pommes, en retirer les pépins, et les couper en petits morceaux
2. Faire cuire les morceaux de pommes à la poêle avec 50 g de sucre et un peu de cannelle
3. Faire préchauffer votre four à 180 °C et beurrer un plat
4. Mélanger dans un saladier 150g de farine, 100g de sucre et 75g de beurre pour obtenir un mélange sableux
5. Mettre les pommes dans le plat et verser le mélange par-dessus, et laisser cuire au four pendant 30 minutes.

UNE RECETTE DE CUISINE

1. Eplucher les 6 pommes, en retirer les pépins, et les couper en petits morceaux
2. Faire cuire les morceaux de pommes à la poêle avec 50 g de sucre et un peu de cannelle
3. Faire préchauffer votre four à 180 °C et beurrer un plat
4. Mélanger dans un saladier 150g de farine, 100g de sucre et 75g de beurre pour obtenir un mélange sableux
5. Mettre les pommes dans le plat et verser le mélange par-dessus, et laisser cuire au four pendant 30 minutes.

ALGORITHMES ET PROGRAMMES

- Une **description précise** d'une méthode de résolution
- Une suite de **tâches élémentaires** connues de l'utilisateur qui va devoir les effectuer
- Caractéristiques requises :
 - Non-ambiguïté : il n'y a aucune initiative à prendre
 - Exhaustivité : tous les cas de figure sont prévus
 - Terminaison : on a toujours un résultat
- Un **programme**, c'est la traduction d'un algorithme dans un langage compréhensible par une machine

OCAML ET PSEUDO-CODE

ÉCRIRE UN ALGORITHME

- Solution universelle : le **pseudo-code**

- Exemple :

Pour chaque pomme,

 Faire

 Eplucher (pomme)

 RetirerPépins (pomme)

 CouperEnMorceaux (pomme)

 Fin faire

- Avantage : on peut ensuite adapter (assez) facilement ce pseudo-code à un langage de programmation précis

UN LANGAGE PRÉCIS : CAML



- Caml : langage de programmation de haut niveau
- Développé et distribué par l'INRIA (Institut National de Recherche en Informatique et en Automatique),
- Typage fort :
 - un code plus propre et plus sûr
 - moins de souplesse pour le programmeur

LE SYSTÈME CORRESPONDANT : OCAML

- Caml et OCaml ?
 - Caml est un langage (une façon d'écrire du code)
 - OCaml est une implémentation de ce langage (qui permet à l'ordinateur de comprendre le code écrit en Caml)
- Alternative : Caml Light (utilisé dans certaines prépas)
 - Version plus légère
 - Pas d' "évaluation" au fur et à mesure avec Emacs
 - Il est conseillé d'utiliser son successeur : OCaml

VARIABLES ET TYPE

QU'EST-CE QU'UNE VARIABLE ?

- Une **variable** est un objet stockée par l'ordinateur
- Il peut s'agir d'un nombre, d'une chaîne de caractère, ou d'une structure de données quelconque (ex : vecteur)
- Désignée par son **nom** :
 - Choisi si possible intelligemment
 - Eviter les espaces, les accents et les caractères spéciaux
 - Utiliser éventuellement des "traits bas" (underscore) (`par_ex`)
 - Jamais deux variables avec le même nom au même moment !

DÉCLARATION DE VARIABLE

- Une **déclaration** de variable consiste à indiquer à l'ordinateur la création d'une variable pour qu'il prenne les mesures nécessaires (notamment l'allocation de mémoire)

- Pseudo-code :

age_du_capitaine ← 42 (nom pas encore utilisé)

- Caml :

`let age_du_capitaine = 42 ;;` (global)

`let age_du_capitaine = 42 in ... ;;` (local)

ACCÈS À UNE VARIABLE

- Pour **accéder** au contenu d'une variable, il suffit d'utiliser son nom, et l'ordinateur se chargera de récupérer la valeur correspondante.
- Pseudo-code :
`afficher(age_du_capitaine)` (affiche 42)
- Caml :
`print_int(age_du_capitaine);;` (affiche 42)

AFFECTATION D'UNE VALEUR À UNE VARIABLE

- L'**affectation** consiste à donner une nouvelle valeur à une variable
- Pseudo-code :
 `age_du_capitaine ← 43` (nom déjà existant)
- Caml :
 - Fausse solution : « écraser » la variable
 `let age_du_capitaine = 43 ;;`
 - Vraie solution : utiliser une référence

COMBINAISONS

- Il est possible de combiner ces opérations :

```
let age_du_capitaine = 42 ;;  
print_int(age_du_capitaine);; (affiche 42)
```

```
let nouvel_age_du_capitaine = age_du_capitaine + 1 ;;  
print_int(nouvel_age_du_capitaine);; (affiche 43)
```

QU'EST-CE QUE LE TYPAGE ?



\neq



QU'EST-CE QUE LE TYPAGE ?

42 \neq "bonjour"

QU'EST-CE QUE LE TYPAGE ?

42 \neq "quarante
deux"

QU'EST-CE QUE LE TYPAGE ?

42 \neq 42,0

LES TYPES LES PLUS CLASSIQUES

- Dès sa déclaration, chaque variable se voit affecter un **type** précis, qui caractérise le type d'information qu'elle contient
- Les types les plus classiques sont :
 - `int` : entier relatif (positif ou négatif) (ex : `-13`)
 - `float` : nombre réel (avec virgule) (ex : `3.14`)
 - `char` : caractère (noté entre apostrophes) (ex : `'c'`)
 - `string` : chaîne de caractères (ex : `"bonjour"`)
 - `bool` : booléen (ex : `true` ou `a>3`)
 - `unit` : rien (noté `()`)

UN PETIT TEST

Quel est le type de chacun des éléments ci-dessous ?

4 . 0

' 4 '

4 > 0

4

" 4 "

TYPES COMPLEXES

Caml reconnaît également des types plus complexes :

- *n*-uplets : `(1, "ab", true)` de type `int * string * bool`
- Vecteurs : `[| 3 ; 6 |]` de type `int array`
- Listes : `["Hey" ; "Bonjour"]` de type `string list`
- Structures fournies dans les autres modules de Caml
- Tout ce qu'on aura envie de créer

CONSTRUCTIONS CLASSIQUES

ITÉRATION

- Parfois, il est nécessaire d'effectuer plusieurs fois la même action, voire la même suite d'actions : on parle d'**itération**, ou de **boucle**.

Pour chaque pomme,

 Faire

 Eplucher (pomme)

 RetirerPépins (pomme)

 CouperEnMorceaux (pomme)

 Fin faire

BOUCLES

Dans la plupart des langages, il existe deux types de boucles :

- Boucle `for`
 - Exécuter n fois une portion de code, ce nombre n étant déterminé à l'avance
 - Bien adapté aux vecteurs et aux chaînes de caractères
- Boucle `while`
 - Exécuter une portion de code tant qu'une certaine condition n'est pas remplie
 - Bien adapté aux listes et autres structures récursives
 - Risque de boucle infinie (le programme ne s'arrête jamais)

BOUCLE FOR

- Une boucle `for` commence par la déclaration d'un **indice** et de deux **bornes** (de type `int`) entre lesquelles cet indice va progressivement évoluer.
- Après chaque passage dans le corps de boucle, l'indice est **incrémenté** (on lui ajoute 1) ou **décrémenté** (on lui retranche 1).
- Lorsque l'indice atteint la deuxième borne, on exécute une dernière fois le corps de boucle et on passe à la suite du programme

UNE BOUCLE **FOR** EN PSEUDO-CODE

- Formulation propre en pseudo-code :

```
Pour p allant de 1 à 6
```

```
  Faire
```

```
    Eplucher (p)
```

```
    RetirerPépins (p)
```

```
    CouperEnMorceaux (p)
```

```
  Fin faire
```

- Le corps de boucle se situe entre **Faire** et **Fin faire**

UNE BOUCLE **FOR** EN CAML

- La même chose en Caml :

```
for  p = 1 to 6
  do
    eplucher (p) ;
    retirer_pepins (p) ;
    couper_en_morceaux (p)
done;
```

- Le corps de boucle se situe entre **do** et **done;**
- Une boucle **for** est de type **unit** (on ne renvoie rien)

VARIANTE DESCENDANTE

- Une variante avec downto :

```
for  p = 6 downto 1
do
    eplucher(p) ;
    retirer_pepins(p) ;
    couper_en_morceaux(p) ;
done;
```

- Remarque : dans certains autres langages, il est possible de spécifier directement le pas, c'est-à-dire la différence entre un indice et le suivant.

BOUCLE **WHILE**

- Une boucle `while` dépend avant tout d'une **condition**, c'est-à-dire une question fermée (à laquelle on répond par oui ou par non).
- Au début de chaque passage dans la boucle, on vérifie que cette condition est vérifiée :
 - Si c'est le cas, on exécute le corps de boucle et on revient tester la condition
 - Sinon on passe à la suite du programme

UNE BOUCLE **WHILE** EN PSEUDO-CODE

- Formalisation propre en pseudo-code :

Tant que (il reste une pomme à couper)

Faire

$p \leftarrow$ ChoisirUnePomme ()

 Eplucher (p)

 RetirerPépins (p)

 CouperEnMorceaux (p)

Fin faire

- Le corps de boucle se situe entre **Faire** et **Fin faire**

UNE BOUCLE **while** EN CAML

- La même chose en Caml :

```
while (nombre_pommes != 0)
  do
    let p = choisir_une_pomme() in
    eplucher(p);
    retirer_pepins(p);
    couper_en_morceaux(p);
    <diminuer nombre_pommes>
  done;
```

- Le corps de boucle se situe entre **do** et **done;**
- Une boucle **while** est de type **unit** (on ne renvoie rien)

AUTRES EXEMPLES À VENIR

- Cette construction se combine bien avec :
 - des références
 - des listes
 - des structures définies récursivement
- La condition est en fait un **booléen** :
 - soit un test booléen (ex : `prenom = "Bob"` ou `age < 18`)
 - soit une fonction qui renvoie un booléen

PETIT EXERCICE

- **Question** : Pouvez-vous réécrire le programme ci-dessous en utilisant une boucle `for` :

```
indice ← 0
Tant que (indice < 10)
    Faire
        Afficher(indice)
        indice ← indice + 1
    Fin faire
```

- **Réponse** :

```
Pour indice allant de 0 à 9
    Faire
        Afficher(indice)
    Fin faire
```

LES CONDITIONS

- Parfois, plusieurs cas de figures de figures sont possibles, et l'action à entreprendre dépend de la situation :

```
Si (Le feu est rouge)
```

```
  Alors
```

```
    S'arrêter
```

```
  Sinon
```

```
    Passer
```

ÉCRITURE EN PSEUDO-CODE

- Pseudo-code :

```
Si (condition)
```

```
    Alors
```

```
        Instruction1
```

```
    Sinon
```

```
        Instruction2
```

```
Fin si
```

EN CAML : IF ... THEN ... ELSE

- En Caml :

```
if (condition)
  then
    instruction1
  else
    instruction2;
```

- La condition est de type bool
- instruction1 et instruction2 doivent être du même type

SANS LE ELSE

- En pseudo-code :

```
Si (condition)
  Alors
    instruction1
Fin si
```

- En Caml :

```
if (condition)
  then
    instruction1;
```

UN PROBLÈME ?

- Que fait le programme suivant ?

```
if (condition)
    then
        instruction1
    else
        instruction2;
        instruction3;
```

- Et celui-ci ?

```
if (condition)
    then
        instruction1
    else
        instruction2;

instruction3;
```

LA SOLUTION : **BEGIN ... END**

- On utilise `begin ... end`
 `if (condition)`
 `then`
 `instruction1`
 `else`
 `begin`
 `instruction2a;`
 `instruction2b`
 `end;`

 `instruction3;`

IMBRICATIONS

- Il est possible d'utiliser plusieurs conditions les unes dans les autres :

```
if (conditionA)
    then
        instruction1
    else
        begin
            if (conditionB)
                then
                    instruction2
                else
                    instruction3
        end;
```

TROP D'IMBRICATIONS...

- Il ne faut pas abuser des bonnes choses...

```
if (prenom="Barack")
  then
    "Obama"
  else
    if (prenom="Vladimir")
      then
        "Poutine"
      else
        if (prenom="Hu")
          then
            "Jintao"
          else
            "Je ne sais pas"
```

LA SOLUTION EN CAML : **MATCH**

- La même chose en plus propre :

```
match prenom with
| "Barack" -> "Obama"
| "Dimitri" -> "Medvedev"
| "Hu" -> "Jintao"
| _ -> "Je ne sais pas"
```

- On verra tout ça en pratique la semaine prochaine

FONCTIONS

FONCTIONS

- Une fonction, c'est l'**implémentation** d'un algorithme, c'est-à-dire sa traduction dans un langage particulier.
- Elle est caractérisée par :
 - Son **nom**
 - Ses **arguments** : le nombre et le type d'objet qu'elle prend en entrée
 - Son contenu : la suite d'actions qu'elle effectue à partir de ces objets
 - Sa **sortie** : le résultat qu'elle renvoie quand elle a fini
- Définir une fonction permet ensuite d'y faire appel plus loin dans le programme.

UN EXEMPLE EN PSEUDO-CODE

- Fonction d'affichage

```
AfficherCarreNombre (n) =  
    Carre ← n*n  
    Afficher (Carre)  
    PasserALaLigneSuivante ()  
    NeRienRenvoyer ()
```

- Fonction principale

```
Main () =  
    Pour i allant de 1 à 10  
        Faire  
            AfficherNombre (i)  
        Fin faire  
    NeRienRenvoyer ()
```

LE MÊME EXEMPLE EN CAML

- Fonction d'affichage

```
let affiche_carre_nombre n =  
    let carre = n*n in  
    print_int(n);  
    print_newline();  
    ();;
```

- Fonction principale

```
let main () =  
    for i = 1 to 10  
        do  
            affiche_carre_nombre(i)  
        done  
    ();;
```

RÉCURSIVITÉ : L'EXEMPLE DE LA FACTORIELLE

- Définition

$$n! = \text{factorielle}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 \times 2 \times \dots \times n & \text{sinon} \end{cases}$$

- Remarque :

Pour $n > 0$, $\text{factorielle}(n) = n \times \text{factorielle}(n - 1)$

- Pseudo-code :

```
Factorielle(n) =  
    Si (n = 0)  
        Alors  
            Renvoyer 1  
        Sinon  
            Renvoyer n * Factorielle(n-1)  
    Fin si
```

"L'ÉTERNITÉ, C'EST LONG, SURTOUT VERS LA FIN" (W. ALLEN)

Ce qu'il ne faut pas faire

Faire un crumble =

1. Eplucher les 6 pommes, en retirer les pépins, et les couper en petits morceaux
2. Faire cuire les morceaux de pommes à la poêle avec 50 g de sucre et un peu de cannelle
3. **Faire un crumble**
4. Faire préchauffer votre four à 180 °C et beurrer un plat
5. Mélanger dans un saladier 150g de farine, 100g de sucre et 75g de beurre pour obtenir un mélange sableux
6. Mettre les pommes dans le plat et verser le mélange par-dessus, et laisser cuire au four pendant 30 minutes.

FONCTIONS RÉCURSIVES

- Une fonction **récursive** est une fonction qui s'appelle elle-même
- Pour éviter de boucler à l'infini, l'appel récursif doit se faire sur un argument différent de celui de la fonction appelante
- Une fonction récursive doit comporter :
 - Un (ou plusieurs) cas de base
 - Un (ou plusieurs) cas récursif(s)

LES FONCTIONS RÉCURSIVES EN CAML

- En Caml, on précise qu'une fonction est récursive avec le mot clef `rec`

- Exemple :

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

- Si vous oubliez ce mot clef `rec`, Caml refusera de comprendre :

```
let factorielle n =  
  if (n = 0)  
  then 1  
  else n * (factorielle (n-1));;  
      Characters 54-65:  
      else n * (factorielle (n-1));;  
                ^^^^^^^^^^^
```

Error: Unbound value factorielle

ARGUMENTS ET TYPAGE

QU'EST-CE QU'UN ARGUMENT ?

Trois définitions plus ou moins rigoureuses :

- Le truc qu'on "donne à manger" à la fonction
- Un des éléments sur lequel on applique la fonction
- Une variable locale à la fonction dont la valeur n'est pas connue à l'avance

EXEMPLES

- Affichage du carré d'un nombre et passage à la ligne

```
AfficherCarreNombre (n) =  
    Carre ← n*n  
    Afficher (Carre)  
    PasserALaLigneSuivante()  
    NeRienRenvoyer ()
```

- Fonction de Syracuse :

```
Syracuse (n) =  
    Si (n est pair)  
        Alors  
            Renvoyer n/2  
        Sinon  
            Renvoyer 3*n+1  
    Fin si
```

$$Syracuse(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3 \times n + 1 & \text{si } n \text{ est impair} \end{cases}$$

CAML, ARGUMENTS ET TYPAGE

- La grande force de Caml est qu'il est capable de deviner tout seul de quel type sont les différents arguments et la sortie d'une fonction
- Affichage du carré d'un nombre et passage à la ligne :

```
let affiche_carre_nombre n =  
  let carre = n*n in  
  print_int(n);  
  print_newline();  
  ();;
```

```
val affiche_carre_nombre : int -> unit = <fun>
```

- Fonction de Syracuse :

```
let syracuse n =  
  if (n mod 2 = 0)  
  then n/2  
  else 3*n+1;;
```

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

LE TYPE DE LA SORTIE EN CAML

- En pratique, en Caml, le type de la sortie est déterminé par la dernière instruction
- Par conséquent, pour une fonction de type `unit`, il suffit que la dernière instruction soit elle-même de type `unit`
- Ces deux fonctions sont donc équivalentes :

```
let affiche_nombre n =  
  print_int(n);  
  print_newline();  
  ();;
```

```
let affiche_nombre n =  
  print_int(n);  
  print_newline();;
```

CAML, ARGUMENTS ET TYPAGE

- Avantage : cela évite de faire n'importe quoi
- Inconvénient : si vous essayez quand même de faire n'importe quoi, Caml vous envoie balader.
- Exemple :

```
syracuse ("Bob") ; ;  
      ^^^^^^^
```

Error: This expression has type string but is
here used with type int

QUELQUES PETITS EXERCICES POUR FINIR...

- Ecrire les fonctions suivantes (avec ou sans récursivité)
 - Somme des entiers de 1 à n $(1 + 2 + \dots + n)$
 - Somme des entiers de a à b $(a + (a+1) + (a+2) + \dots + (b-1) + b)$
 - Produit des entiers de 1 à n $(1 * 2 * \dots * n)$
 - Factorielle : la même chose, mais `factorielle(0) = 1`
 - Minimum de a et de b
 - (Minimum de a et b, maximum de a et de b)
 - Coefficients binomiaux

$$\binom{n}{p} = \frac{n}{1} \times \frac{n-1}{2} \times \dots \times \frac{n-p+1}{p}$$

PROCHAINE SÉANCE

Mardi 2 Octobre

[TD] BASES DE PROGRAMMATION

