Algorithmes et Programmation Impérative 1

Impressions formatées. Compilation.

Objectifs du TP :

1. découvrir le moyen de produire des affichages formatés
2. apprendre à utiliser le compilateur d'Objective Caml (`ocamlc`) ;
3. récupérer les arguments fournis sur la ligne de commande (tableau `Sys.arg`) ;
4. structurer son code source en déclarations et un appel au programme à exécuter.

1 Impressions formatées

Cette section a pour but de présenter une nouvelle procédure d'impression : `printf`.

1.1 Motivation

Vous connaissez plusieurs procédures d'impressions de données :
- `print_int` : `int → unit` ;
- `print_float` : `float → unit` ;
- `print_char` : `char → unit` ;
- `print_string` : `string → unit` ;
- `print_endline` : `string → unit` ;
- `print_newline` : `unit → unit`.

Chacune de ces fonctions est spécialisée dans l'impression d'un seul type de données.

L'instruction donnée ci-dessous illustre une utilisation de certaines de ces procédures pour afficher la table de multiplication par 7,

```
let table = 7
in
  for i = 1 to 10 do
    print_int i ;
    print_string "␣x␣";
    print_int table ;
    print_string "␣=␣" ;
    print_int (i * table) ;
    print_newline ()
  done
```

et voici l'affichage que son exécution produit

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
```

```
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

L'affichage ainsi produit peut être suffisant, mais il existe des circonstances pour lesquels il ne l'est pas. Par exemple, si on exige un alignement vertical parfait des signes `x` et `=`, ainsi que l'alignement vertical des chiffres des unités des nombres, comme dans ce qui suit.

```
 1 x  7 =   7
 2 x  7 =  14
 3 x  7 =  21
 4 x  7 =  28
 5 x  7 =  35
 6 x  7 =  42
 7 x  7 =  49
 8 x  7 =  56
 9 x  7 =  63
10 x  7 =  70
```

Question 1 Avant de poursuivre, prenez le temps de réfléchir quelques instants pour voir si vous arriveriez à programmer simplement avec les procédures d'impression que vous connaissez l'affichage d'une table de multiplication respectant ce format d'affichage.

1.2 La procédure printf

`printf` est une procédure prédéfinie du langage Objective Caml qui permet d'effectuer des impressions formatées assez élaborées. Nous allons donner une présentation non exhaustive de ce qu'il est possible d'obtenir avec elle.

Pour commencer, cette procédure est définie dans le module `Printf`. Son nom pleinement qualifié est donc `Printf.printf`. Si on veut se dispenser de préciser le module à chaque appel, il suffit d'utiliser la directive

```
open Printf ;;
```

ce que nous ne supposerons pas être fait dans la suite.

Le type de cette procédure est assez délicat à décrire. Regardons ce que nous apprend l'interprète du langage à son sujet :

```
# Printf.printf ;;
- : ('a, out_channel, unit) format -> 'a = <fun>
```

Il n'est pas question ici de présenter en détail le type de cette procédure (et vous ne serez pas interrogé sur ce point). Nous nous contenterons d'explorer quelques exemples d'utilisation.

`printf` ressemble à `print_string` : en effet, on peut utiliser `printf` comme on utilise `print_string` pour imprimer des chaînes de caractères.

```
# print_string "timoleon" ;;
timoleon- : unit = ()
# Printf.printf "timoleon" ;;
timoleon- : unit = ()
# print_string "" ;;
```

```
- : unit = ()
# Printf.printf "" ;;
- : unit = ()
```

On en apprend un peu plus sur les chaînes de caractères : les chaînes de caractères peuvent contenir des caractères non affichables. Ces caractères sont dits spéciaux. Pour les représenter dans une chaîne de caractère littérale, on les fait précéder d'une « contre barre » (ou pour les anglophones « backslash »). Nous ne présentons ici que trois de ces caractères spéciaux.

1. Le caractère \n. Il provoque un passage à la ligne à l'impression.

```
# print_string "timoleon\n" ;;
timoleon
- : unit = ()
# Printf.printf "timoleon\n" ;;
timoleon
- : unit = ()
```

   Il peut être situé n'importe où.

```
# print_string "timo\nleon\n" ;;
timo
leon
- : unit = ()
# Printf.printf "timo\nleon\n" ;;
timo
leon
- : unit = ()
```

2. Le caractère \t. Il provoque une tabulation à l'impression.

```
# print_string "timo\tleon\n" ;;
timo    leon
- : unit = ()
# Printf.printf "timo\tleon\n" ;;
timo    leon
- : unit = ()
# print_string "\t\ttimo\tleon\n" ;;
                timo    leon
- : unit = ()
# Printf.printf "\t\ttimo\tleon\n" ;;
                timo    leon
- : unit = ()
```

3. Et le caractère \\. Il se contente simplement d'imprimer le caractère \.

```
# print_string "\\timo\tleon\n" ;;
\timo   leon
- : unit = ()
# Printf.printf "\\timo\tleon\n" ;;
\timo   leon
- : unit = ()
```

Question 2 Pouvez-vous écrire la procédure prédéfinie `print_endline` en utilisant uniquement `print_string` ?

Pour ces caractères spéciaux, nos deux procédures se ressemblent encore. Alors ? ces deux procédures sont-elles bien équivalentes ?

`printf` n'est pas `print_string` : en effet, nous allons le voir avec trois constats.

1. Nous avons déjà noté que le type de `Printf.printf` est
   $$('a, \texttt{out\_channel}, \texttt{unit})\ \texttt{format} \rightarrow {'a}$$
   alors que celui de `print_string` est
   $$\texttt{string} \rightarrow \texttt{unit}.$$
   Les types n'étant pas les mêmes, les deux procédures doivent différer.
2. Avec `print_string` on peut écrire l'instruction
   ```
   # print_string ("timo"^"leon") ;;
   timoleon- : unit = ()
   ```
   Dans cet exemple, la chaîne passée à `print_string` est exprimée comme une concaténation de deux chaînes.
   Cela est impossible avec `printf` :
   ```
   # Printf.printf ("timo"^"leon") ;;
   This expression has type string but is here used with type
     ('a, out_channel, unit) format =
       ('a, out_channel, unit, unit, unit, unit) format6
   ```
   L'interprète refuse l'expression de concaténation sous le prétexte d'un mauvais type. Et oui, nous l'avons vu tout à l'heure, le type du paramètre de `printf` n'est pas une chaîne de caractères, contrairement à ce que pouvait laisser penser les exemples qui ont précédés.
3. Avec `print_string`, on peut demander l'affichage de la chaîne de caractères `"%d"`.
   ```
   # print_string "%d" ;;
   %d- : unit = ()
   ```
   La chaîne est bien imprimée.
   Mais avec `printf`, on obtient
   ```
   # Printf.printf "%d" ;;
   - : int -> unit = <fun>
   ```
   Le résultat est très surprenant ! On constate qu'on obtient une valeur (fonction) de type $\texttt{int} \rightarrow \texttt{unit}$, et surtout, aucune impression n'a été produite.
   D'après le type obtenu pour l'expression `Printf.printf "%d"`, on doit pouvoir passer un paramètre entier supplémentaire à `printf`. Ce qui se vérifie immédiatement :
   ```
   # Printf.printf "%d"  5 ;;
   5- : unit = ()
   # Printf.printf "%d"  (-15) ;;
   -15- : unit = ()
   ```
   Et il n'est pas difficile de comprendre que `Printf.printf "%d"` est équivalent à `print_int`.

`printf` est plus puissant que `print_string` : En fait le caractère `%` dans la chaîne de format qui est passée à `printf` est un caractère spécial qui, accompagné d'information(s) supplémentaire(s)le suivant, indique à `printf` qu'il doit être remplacé par les données qui suivent la chaîne.

Nous allons donner ici quelques exemples de types de données qu'il est possible d'insérer dans une chaîne de format grâce au caractère `%`.

- `%d` : permet d'insérer un nombre entier.
- `%f` : permet d'insérer un nombre flottant.
- `%B` : permet d'insérer un booléen.
- `%c` : permet d'insérer un caractère.
- `%s` : permet d'insérer une chaîne de caractères.

Question 3 Testez chacun de ces formats.

Question 4 Réalisez toutes les procédures `print_XXX` que vous connaissez à l'aide de la procédure `printf` uniquement.

Un exemple de format complexe : Bien entendu, tous les formats présentés peuvent être utilisés dans la même chaîne de format. En voici un exemple :

```
# let article = "tomates"
  and prix_unitaire = 2.80
  and quantite = 1.750
  in
    Printf.printf
      "%.3f␣kilos␣de␣%s␣a␣%.2f␣euros␣le␣kilo␣font␣:%6.2f␣euros\n"
      quantite
      article
      prix_unitaire
      (prix_unitaire *. quantite) ;;
1.750 kilos de tomates a 2.80 euros le kilo font :  4.90 euros
- : unit = ()
```

Question 5 Vous pouvez remarquer que le format `%f` est ici accompagné de nombres (`%.3f`, `%.2f` et `%6.2f`). En modifiant ces valeurs numériques trouvez une interprétation de leur rôle.

Question 6 Testez sur l'exemple de votre choix le format `%`n`d`, où n est l'entier de votre choix.

1.3 Table de multiplication avec `printf`

Nous sommes maintenant en mesure de produire l'affichage d'une table de multiplication avec le format souhaité.

```
let table = 7
in
  for i = 1 to 10 do
    Printf.printf "%2d␣x␣%2d␣=␣%3d\n" i table (i * table)
  done
```

Question 7 Réalisez une procédure nommée `imprimer_table_mult` de type `int` $\rightarrow$ `unit` qui produit un affichage formaté de la table de multiplication par l'entier passé en paramètre.

Puis utilisez votre procédure pour obtenir l'affichage de diverses tables de multiplication.

Question 8 Est-ce que votre procédure d'impression respecte les règles de formatage pour toutes les tables ?

1.4 Une table de logarithmes

Le logarithme népérien (ou naturel) est donné en CAML par la fonction `log` de type `float → float`.

Question 9 Réalisez une procédure qui prend trois paramètres
- a et b de type `float`
- et n de type `int`,

qui produit l'affichage des valeurs de $\ln x$ pour tous les $x \in [a, b]$ par pas de $\frac{(b-a)}{n}$.

Voici à titre d'exemple, l'affichage à obtenir lorsque $a = 1$, $b = 2$ et $n = 10$.

```
  x | log(x)
1.000 | 0.00000
1.100 | 0.09531
1.200 | 0.18232
1.300 | 0.26236
1.400 | 0.33647
1.500 | 0.40547
1.600 | 0.47000
1.700 | 0.53063
1.800 | 0.58779
1.900 | 0.64185
2.000 | 0.69315
```

2 Compilation

Vous avez utilisé jusqu'à maintenant un interpréteur CAML : un interpréteur est un programme interactif qui attend des commandes/instructions en entrée et calcule/réalise directement les expressions/actions correspondantes en modifiant son état interne. Cet aspect interactif est intéressant en phase d'apprentissage d'un langage ou de développement/prototypage d'un programme. Nous voulons dorénavant produire des programmes directement exécutables sans passer par l'intermédiaire d'un interpréteur.

2.1 Qu'est-ce que la compilation ?

En informatique, la *compilation* désigne le processus de transformation d'un programme écrit dans un langage lisible par un humain en un programme exécutable par une machine.

De manière plus générale, il s'agit de traduire un programme écrit dans un langage source en un programme écrit dans un langage cible. Dans notre cas, le langage source est le langage CAML, le langage cible peut-être soit le langage d'instructions d'une machine virtuelle (bytecode), soit le langage d'instructions du processeur de la machine physique (code natif).

Les étapes d'une telle transformation sont représentées à la figure **??**.

Nous n'entrerons pas dans le détail de ces étapes, nous nous intéressons seulement à la manière de produire le résultat, c'est-à-dire de produire un programme objet. Sachez que tous les programmes que vous utilisez sur un ordinateur sont le résultat d'une compilation : les navigateurs,

les éditeurs de texte, les gestionnaires de fenêtres, les systèmes d'exploitation, ... et les compilateurs eux-mêmes.

2.2 Le compilateur `ocamlc`

Le terme *compilateur* désigne un programme qui réalise une compilation : il reçoit en argument le nom d'un fichier contenant un programme à *compiler*, transforme ce programme, et produit comme résultat un fichier contenant le programme en langage cible. Il existe ainsi au moins autant de compilateurs qu'il y a de langages de programmation.

Nous utliserons le compilateur `ocamlc` : `ocamlc` produit du bytecode, c'est à dire des instructions compréhensibles par une machine virtuelle (`ocamlrun` en l'occurence). L'intérêt de produire du bytecode est qu'il est exécutable sur toute machine/processeur disposant de la machine virtuelle, et ce de manière transparente. Nous allons illustrer l'utilisation du compilateur `ocamlc` avec le programme d'affichage de la table de multiplication par 7.

1. créer un fichier `table_mult_7.ml` contenant l'instruction suivante :

```
let table = 7
in
   for i = 1 to 10 do
     Printf.printf "%2d x %2d = %3d\n" i table (i * table)
   done
```

2. dans un terminal, appeler le compilateur en donnant comme argument le nom du fichier à compiler. L'option `-o` permet de spécifier le nom du programme exécutable à générer (`table_mult_7` ici, sinon ce sera `a.out` par défaut) :

```
> ocamlc table_mult_7.ml -o table_mult_7
```

   Si il n'y a pas d'erreur dans le fichier à compiler, le compilateur vous rend la main sans rien afficher. Il aura cependant créé trois fichiers, `table_mult_7.cmi`, `table_mult_7.cmo` et `table_mult_7`. Un listage de votre répertoire de travail permet de le confirmer :

```
> ls -l table_mult_7*
-rwxr-xr-x. 1 levaire lock 46287 fevr.  4 10:45 table_mult_7
-rw-r--r--. 1 levaire lock   224 fevr.  4 10:45 table_mult_7.cmi
-rw-r--r--. 1 levaire lock   403 fevr.  4 10:45 table_mult_7.cmo
-rw-r--r--. 1 levaire lock   107 fevr.  4 10:35 table_mult_7.ml
```

   Les deux premiers fichiers `table_mult_7.cmi` et `table_mult_7.cmo` sont des résultats intermédiaires de la compilation, et nous les utiliserons plus tard avec la compilation séparée. Le fichier exécutable qui nous intéresse est le fichier `table_mult_7`.

3. lancez l'exécution du programme `table_mult_7` depuis ce même terminal :

```
> ./table_mult_7
 1 x  7 =   7
 2 x  7 =  14
 3 x  7 =  21
 4 x  7 =  28
 5 x  7 =  35
 6 x  7 =  42
```

```
 7 x  7 =  49
 8 x  7 =  56
 9 x  7 =  63
10 x  7 =  70
```

Notez que sous Linux, il vous faudra toujours ajouter les caractères `./` devant les noms de vos programmes quand vous souhaiterez les exécuter.

Remarque concernant les systèmes Windows La compilation d'un fichier source CAML sous Windows s'effectue avec la même commande que sous Linux. Elle produit aussi trois fichiers de même nom. Ma différence réside dans l'exécution de l'exécutable produit : elle se fait en invoquant explicitement la machine virtuelle `ocamlrun`.

```
C:\Documents and Settings\Raymond Calbuth> ocamlrun table_mult_7
 1 x  7 =   7
 2 x  7 =  14
 3 x  7 =  21
 4 x  7 =  28
 5 x  7 =  35
 6 x  7 =  42
 7 x  7 =  49
 8 x  7 =  56
 9 x  7 =  63
10 x  7 =  70
```

2.3 Utilité du tableau Sys.argv

Le programme précédent se limite à l'affichage de la table de multiplication par 7. Pour obtenir la table de multiplication par 11, il nous faut écrire un second programme très similaire au précedent où il suffit de remplacer le 7 par un 11. Pour celle par 5, il nous faut ...

Plutôt que de créer et recompiler un programme pour chaque nombre, nous allons créer un seul programme et le paramétrer par un nombre. Le nombre effectif sera passé à l'exécution du programme, de la même manière que l'on passe des paramètres effectifs quand on appelle une fonction. La terminologie est un peu différente, puisque nous nous situons, lors de l'appel à un programme, au niveau du système d'exploitation : on parle d'arguments de la ligne de commande, et non plus de paramètres.

Une ligne de commande désigne une entrée qu'un utilisateur fournit à un interpréteur de commandes. Un interpréteur de commandes est un programme qui réalise l'interface entre un utilisateur et le système d'exploitation. Le programme avec lequel vous interagissez dans un terminal est un interpréteur de commandes.

Un exemple de ligne de commande est la commande de compilation vue précédemment :

```
> ocamlc table_mult_7.ml -o table_mult_7
```

Pour l'interpréteur, une ligne de commande est constituée initialement d'une chaîne de caractères. Il découpe dans un premier temps cette chaîne en morceaux en considérant un ensemble de caractères comme des séparateurs. L'espace est par défaut un séparateur. Chaque morceau devient un argument de la ligne de commande. La ligne de commande de l'exemple est ainsi

constituée des 4 arguments `ocamlc`, `table_mul_7.ml`, `-o` et `table_mul_7`. L'interpréteur considère toujours le premier de ces arguments comme le nom d'un programme exécutable et lance son exécution. De plus il construit un tableau de chaînes de caractères contenant tous les arguments afin que les arguments soient accessibles depuis les programmes exécutables. En CAML, ce tableau des arguments est accessible par l'intermédiaire du module `Sys` sous le nom `Sys.argv`.

Voici en exemple le source en CAML d'un programme qui affiche tous ses arguments :

```
for i = 0 to (Array.length Sys.argv) - 1 do
  Printf.printf "Argument␣%d:␣%s\n" i Sys.argv.(i)
done
```

Question 10 Créez un fichier `print_args.ml` contenant ce programme et compilez le en donnant `print_args` comme nom au fichier exécutable. Exécutez votre programme avec la ligne de commande suivante :

```
> ./print_args a 11 "11" hello world "hello␣world"
```

Que pouvez-vous en déduire quant à l'utilisation des guillemets dans une ligne de commande ?

Question 11 Écrivez un programme qui affiche la table de multiplication d'un nombre qui lui est passé en argument.

Attention au type des éléments du tableau `Sys.argv` : ce sont des chaînes de caractères. Utilisez la fonction de conversion `int_of_string` pour obtenir un entier à partir de l'argument.

- Que se passe-t'il si vous appelez votre programme avec un argument qui ne représente pas un entier ?
- Que se passe-t'il si vous appelez votre programme sans lui donner d'argument ?
- Que se passe-t'il si vous appelez votre programme avec 2 arguments représentant des entiers ?

2.4 Structure d'un programme destiné à la compilation

Un programme destiné à la compilation aura la structure suivante :

- définition des différentes valeurs utilisées dans le programme (variables, fonctions/procédures, ...) ;
- définition des variables globales utilisées dans le programme ;
- programme principal : c'est la séquence d'instructions qui sera exécutée lors de l'appel au programme depuis l'interpréteur de commandes. Si le programme principal est vide, votre programme exécutable ne fera rien ! Sauf définir des fonctions qui ne sont jamais utilisées.

Il faut enfin prendre soin à la validité des arguments, et rappeler le cas échéant à l'utilisateur comment utiliser votre programme exécutable. La vérification de la validité des arguments se fait toujours au tout début du programme principal. On a l'habitude de définir dans chaque programme une procédure de nom `usage` : cette procédure rappelle comment s'utilise le programme et met fin à celui-ci.

```
(*
  auteur : J.L. Levaire
  date   : fevrier 2010
  objet  : affichage d'une table de multiplication
*)

(* Definition des fonctions et procedures *)
let table_mult n =
```

```
  for i = 1 to 10 do
     Printf.printf "%2d x %2d = %3d\n" i n (i * n)
   done

(* Usage *)
let  usage () =
  Printf.printf "Usage: %s n: table de multiplication d'un entier n\n" Sys.argv.(0);
  exit 1

(* Programme principal *)
let principal () =
  (* Verification du nombre d'arguments *)
  if (Array.length Sys.argv) <> 2 then
    usage ()
  else begin
    (* Appel a la procedure table_mult avec conversion de l'argument *)
    table_mult (int_of_string (Sys.argv.(1))) ;
    exit 0
  end

(* appel a la procedure principal *)
let _ = principal ()
```

Question 12 En reprenant les fichiers du TP2, écrivez un programme exécutable de nom `chemin` qui accepte en premier argument une chaîne de caractères représentant un chemin, en second argument une longueur de segment et affiche le chemin correspondant dans la fenêtre graphique en prenant le centre de cette fenêtere comme point initial.

L'utilisation de la bibliothèque graphique `Graphics` nécessite de rajouter lors de la compilation le fichier `graphics.cma` contenant l'archive de la bibliothèque (le placer en première position).

```
> ocamlc -o chemin graphics.cma chemin.ml
```

Remplacez `graphics.cma` par `camlgraph.cma` si vous utilisez la bibliothèque `Camlgraph`, et mettez les deux si vous utilisez les deux (à vérifier).

De plus, pour que la fenêtre graphique ne se ferme pas immédiatemment après le tracé du chemin, placer un appel à la procédure `Graphics.read_key ()` à la fin de votre programme principal. Cette procédure attend la frappe d'une touche au clavier et retourne le caractère correspondant.

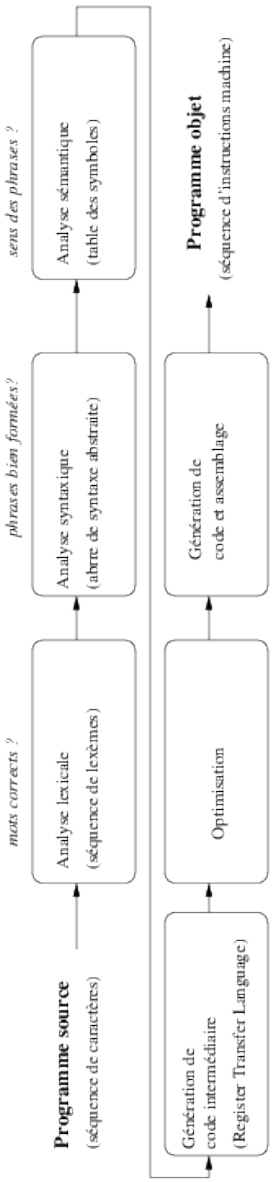

FIGURE 1 – Les étapes d'une compilation.