TP 1 : Initiation à l'environnement Caml (CPP – 1re année)

Février 2006

Ce premier TP a pour but de vous initier à l'utilisation de l'environnement Caml sous Windows, à illustrer les notions de base vues en cours et à apporter quelques compléments pratiques, notamment au sujet de la manipulation de chaînes de caractères.

1 Présentation de l'environnement OCaml pour Windows et méthodologie

Remarque : ceux qui désirent installer l'environnement OCaml sur leur(s) machine(s) personnelle(s), peuvent le trouver sur la page `http://caml.inria.fr/download.fr.html`. Crimson Editor est lui aussi librement disponible, mais sur le site `http://www.crimsoneditor.com`.

Enfin, après l'installation de OCaml, un fichier d'initialisation *.ocamlinit* doit être placé dans le dossier *Objective Caml*. Ce fichier est disponible sur le bureau virtuel `http://grenoble.bv.rhone-alpes.fr/personal/mohrr_inpg/cpp_enseignants/` dans la bibliothèque *Documents Info-CPP/Autres*.

On peut accéder à de la documentation sur OCaml à partir de `http://caml.inria.fr/ocaml/release.fr.html`.

1.1 Prise en main

Pour commencer, lancer l'application OCaml (Menu *Démarrer*, *Programmes*, *Objective Caml* puis *Objective Caml*). L'environnement Caml vous présente alors deux fenêtres[1] :

- la fenêtre principale qui permet de saisir des phrases pouvant s'étendre sur plusieurs lignes (la fin d'une ligne ne se confond pas avec la fin d'une phrase). Les différents mots d'une phrase peuvent être séparés par un ou plusieurs espaces. La fin de la phrase est indiquée par deux points-virgules accolés (`;;`) et est suivie d'un retour chariot (touche « Entrée ») pour la soumettre au système Caml. Si un appui sur la touche « Entrée » n'est pas précédé par `;;`, il provoque simplement un retour à la ligne.
  Après chaque ligne saisie, OCaml affiche un résumé de ce que l'on vient de taper (le type et la représentation) ;
- la fenêtre d'entrées/sorties (intitulée Caml graphics) où seront effectués les affichages du programme, et où seront lues les valeurs entrées par l'utilisateur.

La figure 1 montre un dialogue avec le système Caml et le rôle des différentes fenêtres. Remarquez que l'on peut saisir au plus une phrase par ligne.

[1] Si une seule fenêtre s'ouvre, et que le texte *Modifié pour les CPP INPG* n'apparaît pas, le fichier *.ocamlinit* est mal installé.

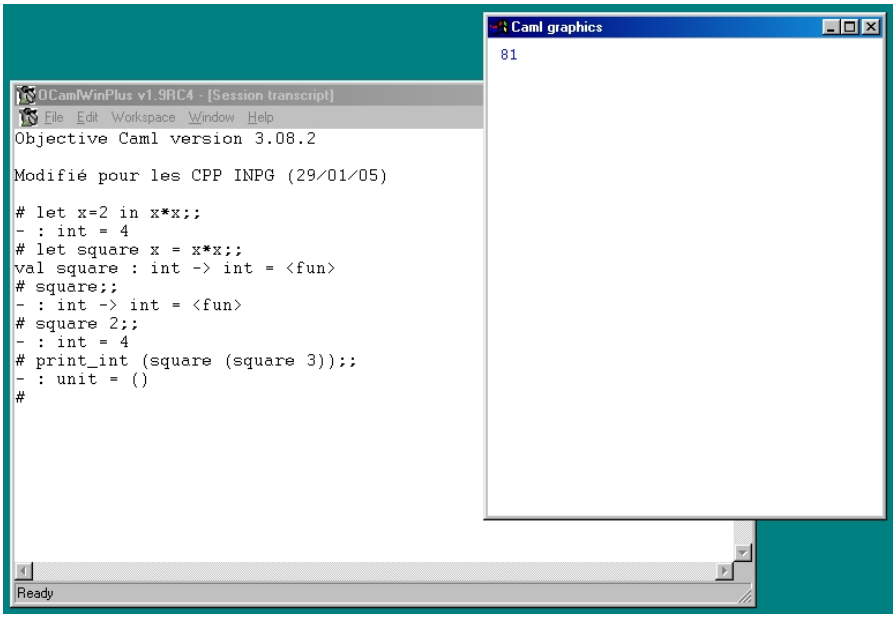

FIG. 1 – Fenêtres de Caml

1.2 Édition

Il est possible de sauvegarder le contenu de l'historique ou de la fenêtre de saisie du terminal dans un fichier en utilisant la commande *Save* du menu *File* (le contenu sauvegardé sera celui de la fenêtre active à ce moment là).

Des fonctions d'édition élémentaires (*copier/couper/coller/imprimer* sont disponibles dans les menus *File* et *Edit*).

Pour des petits programmes de quelques lignes, travailler avec la fenêtre de principale peut s'avérer suffisant. Cependant, lorsque l'on travaille sur des programmes un peu plus volumineux, cette méthode devient vite contraignante. En effet, à chaque fois que l'on souhaite modifier la définition d'une fonction (par exemple pour corriger une erreur que le système Caml nous a signalé), il faut entièrement retaper la fonction, ce qui devient vite pénible. Ce mode d'utilisation à également d'autres inconvénients plus graves :

- lorsque Caml signale une erreur, il est difficile de déterminer quelle est la ligne du programme qui pose problème ;
- lorsqu'on apporte une modification à un programme, on a tendance à modifier directement le texte dans la fenêtre de saisie de l'environnement Caml. Il est très fréquent d'oublier de conserver dans un fichier les changements effectués de cette manière. Finalement, cela aboutit souvent à des versions incohérentes de programmes et à beaucoup de temps perdu.

Pour éviter ces désagréments, nous vous demandons d'**utiliser systématiquement la méthode suivante pour les TP d'informatique** :

- n'écrivez jamais directement vos programmes dans la fenêtre de l'environnement Caml. Pour chaque TP, créez (au moins) un fichier avec l'éditeur « Crimson Editor » (cf. figure 2) : dans le menu *File*, choisir *New* pour créer un nouveau fichier puis *Save as* pour l'enregistrer ;
- pour tester votre programme :
  - enregistrez les dernières modifications de votre fichier avec l'éditeur (menu *File*, commande *Save*) ;

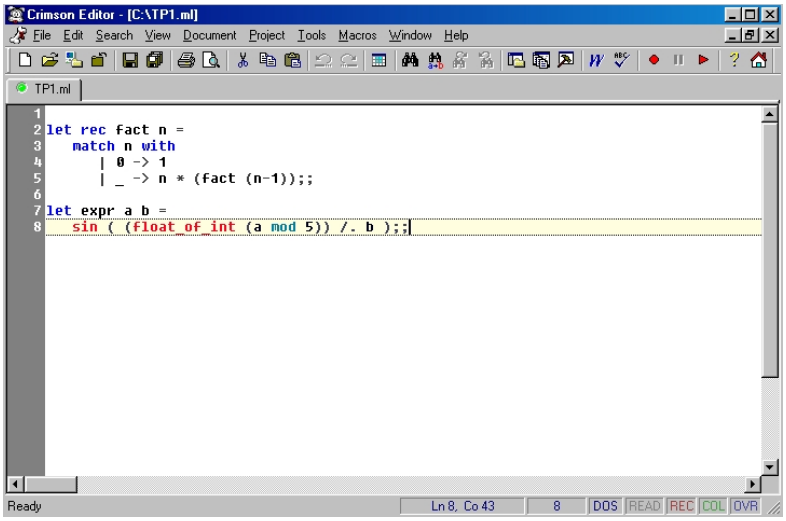

FIG. 2 – Crimson editor

– utilisez la commande *Open* de l'environnement Caml. Celle-ci permet d'envoyer directement le contenu du fichier spécifié au système Caml. La commande *Open* peut être appelée par le biais du menu *File* ou bien l'on peut taper `#use` directement dans la fenêtre de saisie, en précisant le chemin d'accès au fichier concerné, par exemple `#use "D:/Fichiers/test.ml";;`. Si le système Caml détecte une erreur dans une phrase soumise avec la commande *Open*, il indique l'endroit du fichier où se trouve l'erreur (numéros de ligne et de caractère) ;
– on peut ensuite revenir à l'editeur de texte, pour corriger l'erreur. Pour voir les numéros de ligne dans « Crimson Editor », utiliser la commande *Line Numbers* du menu *View*.

Les fichiers utilisés pour sauvegarder des phrases Caml doivent obligatoirement avoir l'extension « .ml ».

On peut donc résumer la méthode de travail de la façon suivante :
– travailler en parallèle avec l'environnement Caml et l'éditeur Crimson Editor ;
– saisir et enregistrer vos fonctions Caml dans un fichier avec l'éditeur ;
– soumettre ce fichier au système Caml avec la commande *Open* ;
– corriger/modifier vos fonctions dans l'éditeur ;
– et ainsi de suite...

Par ailleurs, nous vous recommandons également de conserver une copie de tous les programmes que vous écrirez lors des séances de TP : vous aurez un compte-rendu à rendre pour chaque TP à partir de la prochaine séance et ces programmes pourront également vous aider lors de vos révisions pour l'examen de fin d'année. Conservez toujours (au moins) deux copies de vos fichiers : une sur votre compte informatique et une sur disquette (ou clé USB pour ceux qui sont à la mode) : les incidents matériels sont fréquents et il vaut mieux prévenir que perdre le fruit de quatre heures de travail !

1.3 Spécifications, documentation, commentaires et lisibilité d'un programme

Lorsque l'on écrit un programme informatique, on s'attend à ce qu'il effectue correctement un certain nombre d'opérations. Pour cela, il est important d'établir clairement les spécifications d'un programme — c'est-à-dire ce qu'on attend de lui dans tous les cas de figure possibles — avant de l'écrire. Pour l'instant, l'importance de cet aspect ne vous a peut-être pas frappé car les fonctions vues en cours et TD sont encore relativement simples, mais s'interroger par exemple sur l'intervalle de définition d'une fonction est déjà se soucier des spécifications.

Un autre aspect important (et peut être moins évident à première vue) est celui de la documentation d'un programme. Il est bien sûr important de fournir un mode d'emploi pour les utilisateurs d'un programme (vous n'en aurez pas besoin car vous serez à la fois les concepteurs et utilisateurs de vos programmes) mais il est également important de bien expliquer comment un programme fonctionne, c'est-à-dire le principe de ses algorithmes. En général, on est souvent amené à faire évoluer un programme : pour corriger des erreurs, pour améliorer ses performances (en changeant d'algorithme(s)), pour étendre ses fonctionnalités... Et l'expérience montre qu'il est très difficile de comprendre le fonctionnement d'un programme dont le code n'a pas été documenté. C'est bien sûr le cas lorsque le programme concerné a été écrit par quelqu'un d'autre mais cela se produit tout aussi souvent avec du code que l'on a écrit soi-même et que l'on reprend après un certain temps ! Pour remédier à ce problème, voici deux solutions complémentaires :

- utiliser des noms explicites pour les fonctions, paramètres, variables...
- commenter vos programmes. Il ne faut pas obligatoirement commenter la moindre ligne de code mais au moins expliquer brièvement le but de chaque fonction, ses éventuelles conditions de bon fonctionnement et le principe de l'algorithme utilisé, s'il n'est pas évident. En Caml, on utilise `(*` pour débuter une zone de commentaires et `*)` pour la terminer. Une zone de commentaires peut être insérée n'importe où dans du code, contenir n'importe quel texte (y compris d'autres commentaires) et s'étendre sur plusieurs lignes.

Là encore, les fonctions vues jusqu'à présent ne vous paraissent peut-être pas assez compliquées pour *mériter* d'être commentées mais essayez de garder ces quelques principes à l'esprit lorsque vous écrivez des programmes et de prendre de bonnes habitudes dès ses premiers programmes. Cela devrait vous aider à mieux vous organiser et à atteindre plus facilement (et proprement) votre but. En particulier, cela vous rendra service pour vos révisions et cela aidera aussi les correcteurs de vos TP à comprendre ce que vous avez voulu faire.

2 Premières fonctions

Pour commencer à vous familiariser avec l'environnement Caml, programmez des petites fonctions assez proches de celles vues en cours et en TD. Par exemple :

- une fonction `somme_cubes` prenant deux paramètres entiers et calculant la somme des cubes de ses deux paramètres ;
- une fonction `mult_iter` prenant deux paramètres entiers et calculant la multiplication itérative de ses deux paramètres (cf. TD1) ;
- une fonction `est_pair` déterminant si le nombre entier qu'on lui passe en paramètre est pair ou non. Pour cela, vous pouvez utiliser l'opérateur `mod` qui calcule le reste d'une division entière, ainsi `7 mod 3` donne `1` ;
- deux fonctions `fibo1` et `fibo2`, qui prennent en paramètre un entier n et qui donnent le n^{e} terme de la suite de Fibonacci (elles correspondent aux deux versions dont parle le TD) ;
- une fonction `pgcd` calculant le plus grand commun diviseur (PGCD) de deux nombres entiers. Pour cela, nous vous suggérons d'utiliser l'algorithme d'Euclide. Cet algorithme

part de l'observation que si r est le reste de la division de a par b, alors les diviseurs communs de a et b sont exactement les mêmes que ceux de b et r. D'où l'équation : $\text{PGCD}(a, b) = \text{PGCD}(b, r)$ qui réduit le problème du calcul de PGCD à celui du PGCD de paires de nombres entiers de plus en plus petits. C'est ainsi que $\text{PGCD}(206, 40) = \text{PGCD}(40, 6) = \text{PGCD}(6, 4) = \text{PGCD}(4, 2) = \text{PGCD}(2, 0) = 2$. Il est possible de montrer que pour n'importe quelle paire d'entiers positifs tels que $a > b$, ces réductions aboutissent toujours à une paire dont le second élément est 0, et dont le premier élément est par conséquent le PGCD recherché.

3 Manipulation de caractères et de chaînes de caractères

Les caractères sont désignés par le type `char`. On écrit par exemple `'a' 'b'` ou encore `';'`. Les caractères spéciaux courants sont `'\\'` et `'\''` et `'\r'` et `'\t'`, pour respectivement la contre-oblique, l'apostrophe, le retour-chariot et la tabulation. Les opérateurs de comparaison disponibles sont `=` et `<>` pour l'égalité et la différence. Les opérateurs `>`, `<`, `>=`, `<=` servent à faire des comparaisons sur l'ordre lexicographique.

Les chaînes de caractères sont désignées par le type `string`. Une chaîne doit être encadrée par des guillemets, par exemple abcdef s'écrit `"abcdef"`. Si une chaîne contient le caractère `"`, on doit la faire précéder par une contre-oblique (`\"`) pour ne pas qu'il y ait de confusion avec la fin de la chaîne ; ainsi abc"def s'écrit `"abc\"ef"`. Les opérateurs associés aux chaînes de caractères sont `^` pour la concaténation et `=`, `<>`, `>`, `<`, `>=`, `<=` pour les comparaisons. La bibliothèque standard Caml fournit plusieurs fonctions pour manipuler les chaînes de caractères, dont notamment :

String.length : string -> int renvoie la longueur d'une chaîne ;

String.get : string -> int -> char renvoie le n$^{\text{e}}$ caractère d'une chaîne (la numérotation commence à 0). On peut également utiliser `s.[n]` pour désigner le n$^{\text{e}}$ caractère de la chaîne `s`.

String.set : string -> int -> char -> unit modifie le n$^{\text{e}}$ caractère d'une chaîne ; ce que l'on peut aussi écrire `s.[i]<-c`.

String.sub : string -> int -> int -> string crée une nouvelle chaîne de caractères à partir d'une sous-chaîne de celle passée en paramètre. Par exemple `String.sub "abcdef" 2 3` donne `"cde"` qui commence au 2$^{\text{e}}$ caractère et est de longueur 3.

String.make : int -> char -> string crée une chaîne dont on précise la taille et le caractère de remplissage.

Pour vous entraîner à manipuler les chaînes de caractères, programmez les fonctions suivantes :

- une fonction `est_mon_prenom` qui revoie `true` si, et seulement si, votre prénom est la chaîne passée en paramètre ;
- une fonction `inverse_chaine` renversant une chaîne de caractères.
  Par exemple `inverse_chaine("bonjour")` donne `"ruojnob"` ;
- une fonction `est_palindrome` déterminant si une chaîne de caractères est un palindrome. Un palindrome est un mot qui se lit aussi bien à l'endroit qu'à l'envers, par exemple `"bob"`, `"laval"`, `"serres"`. C'est aussi valable pour les phrases (par exemple `"elu par cette crapule"`) mais nous nous intéressons ici uniquement aux mots car tester les phrases est un peu plus compliqué (il faut ignorer les espaces, les apostrophes, etc.). Nous considérons qu'un mot réduit à un seul caractère est également un palindrome et que c'est également valable pour un mot « vide » ;
- enfin, écrire une fonction `rechercher` qui prend deux chaînes de caractères en argument, et qui dit si la première est une sous-chaîne de la seconde.