MPSI, TP d'informatique: familiarisation avec Caml Light

Alexis Saurin

8 janvier 2003

Présentation
Au cours de ces séances de travaux pratiques, on découvrira peu à peu les différents éléments de Caml Light, on les utilisera pour mettre en application les éléments introduits dans le cours d'informatique (techniques de programmation, algorithmique, utilisation de structures de données...) et on introduira en passant certains concepts fondamentaux de l'informatique.

Ce premier TP va essentiellement nous permettre de nous familiariser un peu avec la syntaxe de Caml Light et la manipulation des fonctions.

1 Expressions de base en Caml Light

Pour commencer, on va se familiariser avec certaines de ces constructions en tapant les programmes suivants dans un éditeur de texte puis en les faisant exécuter par Caml Light et en observant le résultat retourné par l'interpréteur Caml Light.

Certains des exemples comportent délibérément des erreurs pour mettre en évidence des incompatibilités entre certains éléments du langage.

Pour cette partie, chaque question consistera d'une part à :

1. taper les textes des programmes suivants ;
2. essayer de prévoir ce que sera le résultat avant de le faire exécuter par l'interpréteur ;
3. observer le texte qui s'affiche sur la ligne suivante en guise de résultat, en particulier regarder les trois éléments constituant la réponse de l'interpréteur : le nom de l'identificateur défini s'il y en a un, la valeur de l'expression et son type (on fera particulièrement attention aux différents types qui apparaissent, cf plus bas) ;
4. si l'exécution provoque une erreur, comprendre le message, voir d'où il provient et corriger le programme en conséquence ;

puis à répondre aux éventuelles questions supplémentaires.

Exemple d'une exécution :

```
alexis@saragosse:~$ camllight
>       Caml Light version 0.74

#let x = 5 + 11;;
x : int = 16
```

Ici, l'interpréteur donne trois éléments en résultat : `x`, le nom de l'identificateur défini par l'expression, son type `int`, ainsi que sa valeur `16`.

dans une expression plus simple :

```
#5 + 11;;
- : int = 16
```

aucun identificateur n'est défini, ce qui est indiqué par le symbole `-`.

1.1 Valeurs de bases

En Caml Light, on va pouvoir écrire un certain nombre d'expressions, qui seront nos programmes. D'autres expressions seront interdites. On va voir peu à peu ce qu'on peut et ce qu'on ne peut pas écrire en Caml Light.

Tout d'abord, on dispose en Caml Light d'un certain nombre de constantes, certaines représentant des entiers (`0, 1, -1, 2, -2, ...`), d'autres représentant des nombres à virgule flottante (ou encore nombres flottants) (`2.5, 3.99e6, -4.8, ...`), ou encore les constantes booléennes (`true, false`), des caractères (`‘a‘, ‘B‘, ‘;‘, ...`) ou des chaînes de caractères (`"bonjour tout le monde!!!"`), ...

Une expression de Caml Light doit toujours se terminer par “ `;;` ”, c'est la manière en Caml Light de terminer une phrase, pensez-y ! Pour lancer l'exécution, passer à la ligne après avoir tapé les “ `;;` ”.

Question 1:
Taper les expressions Caml Light suivantes et observer le résultat rendu par l'interpréteur ; en particulier, identifier dans chaque cas les deux élements de résultats fournis, comme indiqué ci-dessus. Noter chaque type rencontré et chercher une ou deux autres valeurs de ce type que vous donnerez à l'interpréteur.

```
(* les parties de textes entre (* ... *)
sont des commentaires; ils sont là pour
expliquer le sens de certaines parties
du code, mais ne sont pas considérés par
l'interpréteur. En particulier, ce n'est
pas la peine de les recopier dans votre
éditeur *)
1;;
4.5;;
-9e11;;
true;;
`a`;;
"salut!";;
```

Ici, les types sont très simples (ce sont des types de base de Caml Light), mais on verra plus tard (avec les fonctions d'ordre supérieur, les types somme et produit), que le typage peut être très utile pour éviter certaines erreurs de programmation et pour repérer des erreurs avant l'exécution du programme.

Question 2: Différence entre types `int` et `float`
De même qu'auparavant, essayer les deux expressions suivantes :

```
5;;
5.;;
```

Qu'en déduisez vous sur les types en Caml Light ? Comment prévoyez-vous que vont se faire les opérations arithmétiques en Caml Light ?

1.2 Expressions composées : arithmétiques, booléennes et fonctionnelles

On s'intéresse maintenant à la manière de composer des expressions plus compliquées en Caml Light, pour obtenir de véritables programmes.

Question 3: Expressions entières
Taper les expressions Caml Light suivantes :

```
1+2;;
3-11;;
2*4;;
9/2;;
(* quotient de la division euclidienne *)
5 mod 2;;
(* reste de la division euclidienne *)
(15-6)*2+34-9/2;;
(* les priorités des opérations sont
respectées *)
```

On peut également vouloir comparer des entiers :

```
1<2;;
1=3;;
-3>9;;
```

Question 4: Expressions flottantes
Même chose qu'avant avec :

```
1. +. 3.5;;
1.4 *. 9.88;;
1.3 /. 2.9;;
3.14 -. 0.5;;
2. **. 10.;;
10. **. 20.;;
```

Question 5: Erreurs, incompatibilités entre entiers et flottants
Que se passe-t-il avec les expressions suivantes ?

```
1 + 5.5;;
2 **. 10;;
1. /. 0.;;
1/0;;
```

Expliquer les résultats obtenus. Ces messages sont-ils tous des messages d'erreur ? Essayer de comprendre les messages d'erreur quand il y en a une. Tous les messages signalent-ils des erreurs de la même catégorie ?

Les opérations sur les nombres flottants doivent donc contenir un point pour les différencier des opérations entières.

Question 6: Expressions booléennes

```
1<2;;     (* on peut appliquer < à tous *)
3.4<2.5;; (* les types, mais seulement à *)
`a`<`b`;; (* des expressions de même type *)
1<2.5;;
(1+2)<3 or 1.>0.12;;
not 1=2;;
if 1<2 then 3 else 4;;
1 <= 2;;
1 = 2.5;;
if 3*5<16 then 1 else 0;;
```

Question 7: Les paires et les nuplets

```
(true, 2);;
1,2.5;; (* les parenthèses sont inutiles *)
fst (true, 1.5);;
snd ("coucou", "salut");;
`a`, (1, false);;  (* voyez-vous *)
`a`, 1 , false;; (* la différence? *)
```

```
fst (snd 1, (2.5, false));;
fst (snd (1, (2.5, false)));;
```

Pour les deux dernières lignes, l'erreur vient du fait que l'application de fonction (ici de la fonction `snd`) est prioritaire sur la construction de paires, il faut donc mettre des parenthèses...

Observer toujours bien les types qui apparaissent dans les réponses fournies par l'interpréteur et essayer de bien les répertorier.

Question 8: Définitions

De la même manière qu'en mathématiques on peut écrire “soit a le produit des deux nombres 3 et 5”, on peut écrire en Caml Light `let a = 3*5;;`. L'interpréteur Caml Light répondra : `a : int = 15` ce qui indique que l'on a défini un indentificateur `a` de type entier (`int`) et ayant pour valeur `15`. Une fois qu'un identificateur a été défini, on peut l'utiliser dans d'autres expressions.

```
let x = 2 + 5;;
2*x-1;;
y;;
```

Si on cherche à utiliser un identificateur qui n'a pas été défini, on obtient une erreur, de même que dans un DS de maths où l'on n'indique pas ce que représente une inconnue sur laquelle on raisonne...

```
let test = 1<2;;
if test
then print_string "1 est inf à 2"
else print_string "1 est sup ou égal à 2";;
if test then x else 2*x;;
```

Question 9: Définitions locales

On peut également définir un identificateur localement à une expression :

```
let z=1 in z+x;; (* on définit z localement
et on utilise l'identificateur x défini
globalement à la question 8... *)
z;; (* en dehors de l'expression contenant
la définition locale, l'identificateur
n'est pas "lié", on ne peut pas l'utiliser
car il n'est pas connu de l'interpréteur *)
let test = 3 in test + 4;;
test;; (* une définition locale peut cacher
une définition globale, mais seulement là où
elle est définie... *)
```

Question 10: Les fonctions

Caml Light est ce qu'on appelle un langage fonctionnel, c'est-à-dire que les fonctions sont des expressions comme les autres : elles peuvent être passées en arguments d'autres fonctions, renvoyées en résultat... on peut manipuler une fonction en tant que telle et pas seulement via l'application à un argrument

```
fun x -> x;; (* une fonction est donc
une expression comme une autre... *)
let id = fun x -> x;;  (* ... que l'on
peut nommer grâce à la construction let,
comme toute autre expression de Caml light*)
id 1;;    (* la notation de type 'a -> 'a *)
id 2.99;; (* pour la fonction id indique *)
id true;; (* qu'on peut l'appliquer à *)
id id;;   (* n'importe quoi... *)
fun x -> x+1;;
let incremente = fun x -> x+1;;
let incremente_bis x = x+1;;
let incremente_ter = function x -> x+1;;
(* les trois expressions ci-dessus sont
équivalentes, on remarque que du fait de
l'utilisation de l'opérateur +, l'argument
de incremente ou incremente_bis ne peut
être que de type int (même pas de type
float puisque l'opérateur d'addition pour
les flottants est +. et non +) *)
incremente 2;;
incremente 4.5;;
```

2 Premiers programmes en Caml Light

On va maintenant écrire quelques programmes en Caml Light utilisant les expressions que l'on a rencontré jusque là.

2.1 Un peu de calcul

Question 11:

Construire la fonction qui ajoute 3 à un entier.

Construire la fonction qui rend 0 si son entrée vaut 0 et 1 si c'est n'importe quel autre entier.

Question 12: Élévations au carré

Programmer en Caml Light la fonction qui élève au carré, dans le cas d'entiers (`carre_int`), dans le cas de flottants (`carre_float`).

Testez `carre_int` sur quelques valeurs que vous choisirez, puis sur les nombres 262144 et 16777216. Comment vous expliquez-vous ces résultats ?

On dispose des fonctions `int_of_float` et `float_of_int` qui permettent de faire la conversion entre entiers et flottants. Essayez-les sur quelques exemples significatifs pour voir leur comportement.

Que pensez-vous de la définition suivante de `carre_int` :

```
let carre_int x = int_of_float (carre_float
                  (float_of_int x));;
```

? En quoi cette définition est-elle moins satisfaisante que la précédente ? en quoi peut-elle poser problème (pensez au fait que sur un ordinateur, `int` n'est pas vraiment égal à $\mathbb{N}$ et que `float` n'est pas vraiment $\mathbb{R}$...) ? Pose-t-elle vraiment problème ici ?

Question 13: Des calculs erronés

On a vu à la question précédente que `carre_int` avait sur certains nombres un comportement que l'on n'avait pas prévu au départ. Cela est dû aux limites physiques de l'ordinateur et à la représentation interne des entiers (ce n'est pas un trait propre à Caml Light, tous les langages de programmation ont des limitations de ce genre). On va voir maintenant un autre cas où cette limitation apparaît (mais pour les flottants cette fois) :

Construire la fonction `constante_1` qui prend un couple de flottants (a,b) et qui retourne `1.` si a est nul et $\frac{a+b-b}{a}$ sinon. Testez cette fonction sur différentes valeurs du couple (a, b), a restant petit et b devenant grand (notamment : `1000000000000.`, `100000000000000.`, `100000000000000000.` pour b, a valant `1.`).

Question 14: Des définitions par récurrences ?

Les fonctions numériques qu'on a défini jusqu'ici ne sont pas très intéressantes. Cela vient en grande partie du fait que la plupart des fonctions calculables intéressantes de $\mathbb{N}$ dans $\mathbb{N}$ sont définies par récurrence (ou au moins contiennent quelque part dans leur définition une définition par récurrence), et qu'on ne sait pas pour l'instant définir en Caml Light une fonction par récurrence.

Essayer le texte suivant dans l'interpréteur Caml Light :

```
let f x = if x=0 then 1 else f (x-1);;
```

puis :

```
let f n = 2*n;;
let f x = if x=0 then 1 else f (x-1);;
f 4;;
```

Comprenez-vous ce qui se passe ?

On verra les définitions récursives dans un prochain TP...

2.2 Manipulation de fonctions

Puisque pour l'instant on dispose de peu de choses pour définir des fonctions numériques, on va plutôt s'intéresser à la notion de fonctionnalité en elle-même : on va s'intéresser à des fonctions manipulant d'autres fonctions.

Question 15: Fonctions d'ordre supérieur

1. Construire une fonction `applique` qui prend un couple constitué d'une fonction f et d'un élément x en argument et qui retourne f(x). Observez bien le type de la fonction `applique`.
2. Construire la fonction `compose` qui prend une fonction f et une fonction g en arguments et qui retourne la fonction composée : fog.

Question 16: Fonctions à plusieurs arguments, curryfication et décurryfication

1. Définir une fonction `somme1` qui prend en argument un entier n et retourne la fonction qui ajoute n ;
2. Définir une fonction `somme2` qui prend en argument une paire d'entiers et retourne leur somme.

Dans les deux cas précédents, on a en fait cherché à construire une fonction de plusieurs arguments de deux manières différentes :
- la première manière : `let f x y = ...` est appelée le style curryfié ;
- la seconde manière : `let f (x,y) = ...` est appelée le style décurryfié ;

On voit bien que l'on peut exprimer exactement les mêmes fonctions dans chacun des deux styles[1], même si syntaxiquement, les deux styles sont différents (il suffit de regarder les types de `somme1` et `somme2`...).

On veut rendre automatique la traduction d'un style dans l'autre. Écrire une fonction `curryfier` et une fonction `decurryfier` qui font cette traduction.

[1] cela vient en fait de l'isomorphisme entre les espaces de fonctions C^{A*B} et $(C^B)^A$