Informatique Option
Premiers pas en Caml Light

Sujet (l' Victor Nicollet 2006, 2007)

12 mars 2008

```
#1;;
- : int = 1
#2+2;;
- : int = 4
#sin (0.);;
- : float = 0.0
#
```

Le terminal attend de rencontrer le mot-clé `;;` (deux point-virgule consécutifs) avant de traiter le programme.

La boîte ci-contre décrit un exemple de résultat: la sortie du terminal affiche le code qui a été envoyé et exécuté (les lignes commençant par le symbole dièse), suivi du type du résultat et de sa valeur.

0.1 Types des expressions

Le terminal donne le type de chaque résultat: `int` pour un entier, `float` pour un réel, `bool` pour une valeur booléenne, `string` pour du texte. De plus, il interdit l'exécution d'une opération sur des objets qui ne sont pas du bon type.

```
(1 +  2) = 3;;
 1 + (2  = 3);;
"Texte" + "autre texte";;
```

```
exp(1.234) < 1.5;;
2 * 3.141592;;
```

Question 1 *Examinez la liste de programmes donnée ci-contre: à votre avis, quel sera le type du résultat de chacun ? Lesquels comportent une erreur ? Pourquoi cette erreur a-t-elle lieu ? Comment la corriger ?*

0.2 Noms de variables

Au moyen de l'instruction `let`, il est possible de donner un nom à une valeur par exemple le nom `pi` à la valeur 3.141592. Lorsqu'il est utilisé, un nom est toujours immédiatement remplacé par la valeur qui lui est associé.

```
#let pi = 3.141592;;
pi : float = 3.141592
```

Lorsqu'une telle instruction est exécutée par le terminal , elle renvoie une réponse de la forme suivante:

nom : *type* = *valeur*

Un nom continue à designer la valeur à laquelle il est attaché jusqu'à ce qu'une nouvelle instruction `let` lui indique de désigner une autre valeur. Cette modification n'a aucun effet sur les utilisations précédentes du nom (puisque celles-ci ont été remplacées immédiatement par la valeur associée).

```
let x = 1;;
let y = x + x;;
let x = 2;;
x + y;;
```

Question 2 *Essayer de prévoir le résultat de la suite d'instructions ci-contre sans l'exécuter, puis vérifier les prédictions à l'aide du terminal.*

Il est important de bien choisir les noms donnés aux variables: `a`, `x` ou `bool` sont des noms qui n'apportent pas en général d'information pertinente à un lecteur humain. Au contraire, des noms comme `count`, `discriminant` ou `found` donnent des informations sur l'algorithme utilisé. Il est en général préférable d'écrire plus pour avoir un programme plus lisible.

1 Outils fréquemment utilisés

1.1 Fonctions

```
let carre = function x → x * x;;
carre(2);;

let carre(x) = x * x;;
carre(2);;
```

En plus des valeurs de base vues précedemment, permet de manipuler des fonctions. Par exemple, on peut définir une fonction qui associe à chaque entier son carré. Comme pour une fonction mathématique, son type est `int → int`. Les deux définitions ci-contre sont équivalente (on utilisera la seconde, plus courte).

Question 3 *Reprendre la fonction* `carre` *ci-dessus, et l'utiliser pour définir la fonction polynôme* $P(X) = X^2 + 2X + 1$. *Calculer* $P(1)$ *et* $P(2)$.

1.2 Conditions

```
let abs(x) =
  if x < 0
  then -x
  else x;;
```

Lorsque l'expression à évaluer doit dépendre de la valeur d'une autre expression, on utilise la structure conditionnelle `if then else`. L'exemple ci-contre définit la fonction valeur absolue en utilisant la position de x par rapport à zéro.

Question 4 *Écrire la fonction définie par:*

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

Note: `a mod b` *est égal à zéro si et seulement si* `b` *divise* `a`.

1.3 Définitions locales

```
let cube(x) =
  let par_x(y) = y * x in
  par_x(par_x(x));;
```

Parfois, il peut être utile de faire dépendre une définition de la valeur d'une autre variable, par exemple de l'argument d'une fonction. Le code ci-contre définit, à l'intérieur de la fonction `cube`, une fonction `par`$\backslash_x$ qui multiple son argument par `x`.

Le nom d'une variable définie localement est associée à sa valeur lorsque le `in` correspondant est atteint (et pas avant). Cette association disparaît à la fin de la fonction englobante (il est impossible d'utiliser `par`$\backslash_x$ à l'extérieur de `cube`).

On remarque au passage qu'un argument de fonction est une variable locale définie automatiquement par .

```
let f(x) =
  let x = x - 2 in
  let f(x) = x * x - 1 in
  let f(x) = f(x+3) in
  let f = f(x) in
  f;;
```

Question 5 *Déterminer toutes les racines du polynôme* f *défini ci-contre. On pourra essayer de déterminer, pour chaque variable, l'endroit où elle est définie.*

La question ci-dessus illustre qu'il ne faut pas abuser des définitions locales (ni leur donner des noms incompréhensibles ou trop ressemblants). Une définition locale doit simplifier le code.

1.4 Couples et n-uplets

Un n-uplet est une suite de n valeurs séparées par des virgules. N'importe quelle valeur (entier, fonction, p-uplet etc) peut être élément d'un n-uplet.

```
let couple = (1,2);;
let (x,y) = couple;;
```

```
let triplet = (x,y,3);;
let f(x,y,z) = x + y + z;;

f(1,2,3) = f(triplet);;
```

La première expression permet de nommer les n éléments d'un n-uplet, et la deuxième d'utiliser des functions à plusieurs variables pour lesquelles il est possible de passer les arguments un à un ou tous en même temps.

L'exemple ci-contre décrit deux idiomes de : utiliser un `let` pour obtenir les éléments d'un n-uplet, et considérer une fonction à arguments multiples comme une fonction d'un n-uplet.

Question 6 *On représente des nombres complexes comme des couples de réels* `(re,im)`*. Définir des fonctions* `conj(z)` *(conjugué de z) et* `mul(z1,z2)` *(produit de z_1 et z_2). Écrire une fonction qui calcule $|z|^2$ pour tout z.*

2 Sujets avancés

2.1 Propriétés fonctionnelles

```
let poly(a,b,c) =
  function x → (a * x + b) * x + c;;

let p = poly(1,2,1);;
p(1);;
p(2);;
```

Une fonction est un objet comme les autres. Elle peut par exemple être un argument d'une autre fonction, ou même être renvoyée par une autre fonction. L'exemple ci-contre définit une fonction `poly` qui reçoit en argument trois coefficients a, b, c et renvoie la fonction polynôme $aX^2 + bX + c$. Il propose également une solution à la Question 3.

```
let poly(a,b,c)(x) =
  (a * x + b) * x + c;;
```

La notation curryfiée (nommée d'après l'informaticien Haskell Curry, qui en est l'inventeur) offre un raccourci pour définir de telles fonctions (voir ci-contre).

Question 7 *Construire une fonction* `compose`*, qui à f et g associe la fonction $f \circ g(x) = f(g(x))$. En déduire une fonction* $\texttt{trois}_{\backslash fois}$ *qui à f associe $f \circ f \circ f$. En se servant de la fonction* `carre` *définie précédemment, définir une fonction pour mettre à la puissance 8 un entier.*

2.2 Polymorphisme

```
#let id(x) = x;;
id : 'a → 'a = <fun>
#let add1(x,y) = id(x + y)
add1 : int * int → int = <fun>
#let add2(x,y) = id(x) + y
add2 : int * int → int = <fun>
```

Considérons la fonction identité, qui renvoie son argument. Son type peut être lu de la manière suivante: pour tout type `'a`, la fonction est de type `'a → 'a`.

Dans l'exemple ci-contre, l'argument de `id` dans `add1` est de type `int` (puisque c'est le résultat d'une somme) et donc sa valeur de retour est également `int`. Dans `add2`, la valeur de `id(x)` est de type entier (puisque utilisée dans une somme), donc son argument doit également être de type `int`. L'expression `'a → 'a` signifie donc que l'argument et la valeur de retour doivent avoir le même type.

Question 8 *L'opération* `=` *est de type* `'a → 'a → bool`*. En déduire une fonction* `etrange` *dont le type est* `'a * 'b * (int → 'c) → ('c → 'b) → 'a`

2.3 Effets de bord

```
#let f = random__int;;
f : int → int = <fun>
#f(2);;
- : int = 0
#f(2);;
- : int = 1
```

Si f est une fonction mathématique sur les entiers, alors $f(2)$ a toujours la même valeur. En , il est possible d'avoir des *effets de bord*: certaines fonctions peuvent modifier les valeurs des appels de fonctions ultérieurs, ou plus généralement avoir un effet extérieur au programme (afficher du texte pour l'utilisateur, dessiner des figures, jouer des sons).

```
#print_endline "Test";;
Test
- : unit = ()
```

```
#let de1 = random__int(6) + 1;;
de1 : int = 2
#let de2() = random__int(6) + 1;;
de2 : unit → int = <fun>
#de2();;
- : int = 1
#de2();;
- : int = 2

#print_endline "Test"; de2();;
Test
- : int = 5
```

On introduit un nouvel objet: `()`, dont le type est `unit`, et qui représente une absence de valeur. Par exemple, une fonction comme `print\`$_{endline}$ affiche du texte à l'écran, mais ne renvoie pas de valeur utilisable par le programme: on considère que cette valeur est `()`.

L'autre usage de `()` est de servir d'argument-fantôme. Ci-contre, `de1` est défini comme une valeur aléatoire entre 1 et 6 (il aura donc toujours la même valeur), tandis que `de2` est une fonction (qui prend un argument inutile) et qui renvoie une nouvelle valeur aléatoire entre 1 et 6 à chaque appel.

Un point-virgule simple `;` permet de séparer plusieurs opérations: `a;b;c` va exécuter `a`, puis `b`, et enfin renvoyer la valeur de `c`. Il faut que le résultat de `a` et `b` soit de type `unit`.

Question 9 *Écrire une fonction qui prend un complexe dont les parties réele et imaginaire sont prises au hasard dans* $[0;1]$*, l'affiche sous la forme* $z = a + \mathbf{i}b$ *et affiche également son module* $|z|$*.*

On pourra utiliser la fonction `random\_\`$_{float}$`(x)` *pour obtenir un réel aléatoire entre 0 et* `x`*, la fonction* `sqrt(x)` *pour calculer* $\sqrt{x}$*, et les fonctions* `print\`$_{string}$ *et* `print\`$_{float}$ *pour afficher du texte ou une valeur réelle.*

3 Questions difficiles

Question 10 *Soit un programme ne contenant aucune valeur ou fonction polymorphes. Montrer qu'on peut toujours réécrire ce programme de manière à ne pas utiliser* `let`*, sans effectuer de calculs supplémentaires.*

Question 11 *Soit un programme contenant l'opérateur* `;`*. Montrer que ce programme peut être reformulé sans utiliser* `;` *(on autorise, bien entendu, un unique* `;;` *à la fin du programme).*

Question 12 *Soit une fonction* `f(x,y)` *prenant un couple en argument. Montrer qu'il est possible de réécrire cette fonction en forme curryfiée. En déduire une fonction* `curryfie2` *qui prend des fonctions comme* `f` *en argument et renvoie la version curryfiée de ces fonctions.*

4 Bonnes adresses

Il est possible de télécharger à l'adresse suivante:

`http://caml.inria.fr/download.fr.html`

Le manuel de référence en anglais, qui n'est pas vraiment adressé aux débutants en informatique, se trouve quant à lui sur:

`http://caml.inria.fr/pub/docs/manual-caml-light/`