

Examen du 04 septembre 2006

Documents autorisés - Durée 2 heures

Exercice 1 : types et objets

Indiquer ce qu'infère la boucle d'interaction d'O'Caml sur les textes suivants (tous typables) :

1. déclaration de classe :

```
class [ 'a, 'b ] dico () =  
  object  
    val mutable data = []  
    method reset () = data <- []  
    method add ( key : 'a ) (value : 'b ) = data <- (key, value) :: data  
    method get ( key : 'a ) = List.assoc key data  
  end ;;
```

2. création d'instance, appel de méthode et application partielle :

```
let d1 = new dico ();;  
d1#add "Pierre" 18;;  
d1#get "Pierre";;  
d1#reset ();;  
d1#get "Pierre";;  
d1;;  
let x2 = new dico;;  
x2;;  
let d2 = x2 ();;
```

3. type objet et programmation fonctionnelle :

```
let f x ( d : (string , int) #dico ) = try d#get x; d#reset(); d with _ -> d;;  
  
let c = f "Pierre" ;;  
  
c d1;;
```

Exercice 2 : affectation tracée

On cherche sur des valeurs physiquement modifiables à enregistrer les différentes valeurs affectées pour pouvoir le cas échéant revenir à une de ces anciennes valeurs et pouvoir définir des points de choix.

Pour cela on définit les types 'a voutm (permettant de coder une valeur (constructeur V) ou une marque (constructeur M)) et 'a valeur suivants.

```
type 'a voutm = V of 'a | M;;  
type 'a valeur = {mutable c : 'a; mutable trace : 'a voutm list}
```

1. Ecrire une fonction `acces` de type 'a valeur -> 'a qui retourne la valeur du champ `c` du paramètre.
2. Ecrire la fonction `affectation` de type 'a valeur -> 'a -> unit qui conserve dans le champ `trace` l'ancienne valeur du champ `c` avant de lui affecter la valeur passée en deuxième paramètre.

3. Ecrire une fonction `marque` de type `'a valeur -> unit` qui pose une marque dans le champ `trace` du paramètre.
4. Ecrire une fonction `avant1` de type `'a valeur -> unit` qui remet la valeur du champ `c` dans l'état précédent sa dernière affectation et met à jour le champ `trace`.
5. Ecrire une fonction `avantM` de type `'a valeur -> unit` qui remet la valeur du champ `c` dans l'état précédent sa dernière marque et met à jour le champ `trace`.
6. Indiquer après chaque phrase la valeur de la variable `ex` dans l'exemple suivant :

```
let ex = {c = 0; trace = []};;
affectation 1 ex;;
marque ex;;
affectation 2 ex;;
affectation (1 + (accés ex)) ex;;
marque ex;;
affectation 4 ex;;
avant1 ex;;
avantM ex;;
```

7. On cherche à définir une nouvelle structure de contrôle $Choice(l, e_1, e_2)$, appelée point de choix, qui prend deux expressions e_1 et e_2 et une liste de `'a valeur` l . Cette structure de contrôle permet de lancer le calcul de e_1 en marquant les valeurs de l et si l'exception prédéfinie `Backtrack` est déclenchée pendant ce calcul, de revenir à l'état avant marquage des valeurs de la liste l et de lancer le calcul de e_2 . Indiquer comment vous traduiriez le contrôle $Choice(l, e_1, e_2)$ en O'Caml.

Problème : interprétation d'un langage à pile

Le but de ce problème est de réaliser un interprète du langage EXAM, langage à pile à la FORTH ou à la PostScript. Le texte suivant est repris de l'encyclopédie Wikipedia sur FORTH.

langage EXAM : Description

EXAM est un langage à pile simple et extensible. Une de ses importantes caractéristiques est l'utilisation d'une pile de données pour passer des arguments entre les «mots», qui sont les constituants d'un programme EXAM.

Exemple

L'expression $2+3*4$ qui s'écrit en polonaise inversée en `2 3 4 * +` est considérée comme une suite de mots dont l'évaluation agit sur la pile de données de la manière suivante :

- d'empiler successivement les valeurs 2, 3, puis 4
- de remplacer ensuite les 2 nombres du sommet de la pile (3 et 4) par leur produit 12.
- et enfin de remplacer les 2 nombres en haut de pile (2 et 12) par leur somme 14.

Opérateurs

Les commentaires entre parenthèses indiquent l'effet qu'a l'opérateur sur la pile, qui par convention croît vers la droite. Les commentaires de fin de ligne commencent au caractère `\`.

- opérateurs de pile

```
DUP ( n -- n n )      \ dupliquer le sommet de pile
DROP ( n -- )         \ supprimer le sommet de pile
SWAP ( a b -- b a )   \ échanger des deux éléments du sommet
ROT ( a b c -- b c a ) \ rotation des trois éléments du sommet
```

- opérateurs arithmétiques : `*`, `/`, `+`, `-` (`a b -- entier`)
- opérateurs de comparaison : `=`, `<>`, `<`, `>` (`a b -- booléen`)

Définitions

Le programmeur construit le vocabulaire de son application en définissant ses propres mots. La définition d'un nouveau mot suit la grammaire suivante où `liste_de_mots` correspond à une suite possiblement vide de mots :

```
définition      :=      : NOM liste_de_mots ;
liste_de_mots   :=      mot*
```

L'appel à un nouveau mot est identique aux opérateurs prédéfinis. Un mot EXAM est l'équivalent des sous-programmes, fonctions ou procédures dans les autres langages.

```
: CARRE DUP * ;      ( définition de CARRE )
11 CARRE             ( on obtient 121 en sommet de la pile )
: CUBE DUP CARRE * ; ( usage de CARRE dans une définition )
2 CUBE              ( on obtient 8 en sommet de la pile )
```

Exécution

La syntaxe d'EXAM est tellement simple qu'elle nécessite ni analyse syntaxique, ni analyse lexicale. En fait un système EXAM a deux états possibles. L'état par défaut est le mode interactif qui évalue un mot de la manière suivante :

1. Si le mot est un opérateur prédéfini, l'évaluer,
2. sinon chercher la définition du mot dans le dictionnaire ;
3. si le mot est défini récupérer sa définition et l'exécuter,
4. sinon essayer de le convertir en nombre pour le mettre sur la pile.

Le deuxième mode de fonctionnement du système est le mode compilation, qui permet d'ajouter de nouvelles définitions au dictionnaire. On a déjà vu le mot spécial `:`, qui permet de rentrer en mode compilation. Ainsi la phrase `: CARRE DUP * ;` crée une nouvelle entrée `CARRE` dans le dictionnaire et y met le code du corps de la définition (adresses des mots `DUP` et `*`). Le `;` permet de revenir en mode interactif.

Questions

On se donne les types suivants :

```
type valeur = Iv of int | Bv of bool;;
type pile = valeur stack;;
type element =
  Dup | Drop | Rot | Swap | Mul | Div | Add | Sub
| Eq | Neq | Lt | Gt | I of int | B of bool | Mot of string;;
```

1. trace : Soit le mot TUCK suivant : `: TUCK DUP ROT SWAP ;` Indiquer ce qu'il fait en décrivant l'état de la pile `a b c --` avant et après son appel.
2. étape 1 : Ecrire une fonction `evalop` qui prend une pile et un opérateur de pile (`DUP`, `ROT`, `SWAP`, `DROP`) prédéfini, et évalue cet opérateur dans cette pile. On eut utiliser le module `Stack` de la bibliothèque `O'Caml`.
3. étape 2: Compléter la fonction `evalop` pour traiter les opérateurs `Add` et `Lt`.
4. étape 3 : Ecrire une fonction `evalphrase` qui prend une pile, une liste d'éléments et les évalue dans l'ordre en suivant le schéma suivant : si c'est une constante, alors l'empiler, si c'est un opérateur prédéfini l'évaluer, si c'est un mot s'arrêter.
5. étape 4 : Un dictionnaire est une liste d'association dont chaque association contient sous la forme d'un couple : le nom du mot et la liste des éléments de sa définition. Modifier en fonction du type `dico` que vous définirez la fonction `evalphrase` de sorte que si l'on rencontre un mot on cherche sa définition dans le dictionnaire pour l'évaluer en la place en tête de la liste des éléments à évaluer.
6. étape 5 : On modifie de nouveau la fonction d'évaluation pour pouvoir ajouter des définitions au dictionnaire. La syntaxe d'une définition est indiquée plus haut : elle commence par `:"` suivi du nom et se termine par `;"`. Indiquer avant de l'implanter votre choix pour traiter une définition.