Auteurs : R. Namyst, C. Nichitiu, I. Todinca
Enseignants : L. Bougé, A.Bonhomme le 6/11/2000
Email : {Luc.Bougé,Alice.Bonhomme}@ens-lyon.fr

# TD Programmation 5 : Continuations

## 1 Programmer par continuation

### 1.1 Recherche dans une liste

- Définissez un type `ilist` avec ses fonctions de manipulation `empty`, `cons`, `car`, `cdr`, `is_empty`.
- Ecrire une fonction `generate n` qui génère une liste de `n` éléments qui ont une valeur tirée aléatoirement entre `0` et `n`.
- Écrire une fonction récursive de recherche d'un élément dans une liste (`searchList a l`).
- La reprendre dans un style par continuation : `searchList a l k`. Tester cette fonction avec comme continuation l'impression d'un message indiquant si l'élément est dans la liste ou non.
- Écrire les fonctions `append` et `rev` dans un style par continuation et les tester avec des continuations de votre choix.
- Écrire maintenant la fonction `rev` sous forme itérative avec un itérateur `do_while` que vous définirez.
- Gérer l' échappement `break` en passant la continuation correspondante en paramètre au corps de la boucle. Celui-ci est donc une fonction `body break k`. Que pensez-vous de `continue`, ô futurs chercheurs ?

### 1.2 Recherche dans un arbre

- Définissez un type `itree` arbre avec ses ses fonctions de manipulation `empty`, `make`, `get_item`, `get_left`, `get_right`, `is_empty`).
- Ecrire une fonction `generate n` qui génère un arbre de `n` éléments qui ont une valeur tirée aléatoirement entre `0` et `n`.
- Écrire une fonction récursive de recherche d'un élément dans un arbre (`searchTree a t`).

– La reprendre dans un style par continuation : `searchTree a l k`. Tester cette fonction avec comme continuation l'impression d'un message indiquant si l'élément est dans l'arbre ou non.

– Dans cette deuxième fonction de recherche, rajouter l'affichage des éléments parcourus et des appels aux fonction de continuation. Que peut-on en déduire ?

– Reprendre cette fonction en passant en paramètre supplémentaire la continuation d'échappement `break`, c'est-à-dire la continuation de l'appel initial :

  ```
  let search a t k =
    let rec search_aux a break t k =
      ...
    in search_aux a t k k;;
  ```

  Appeler directement `break` lors de la découverte de l'élément recherché. Vérifier l'effet sur l'affichage des éléments parcourus et des appels aux fonction de continuation. Que peut-on en déduire ?

# 2 Calculatrice

**Remarque.** Pour vous aider dans la mise en œuvre, vous trouverez à l'URL

`http://www.ens-lyon.fr/~bouge/Teaching/Programmation/TD/tpcaml-4.ml`

un harnais général pour les entrées-sorties et la boucle d'évaluation (*toplevel*). Il se lance en évaluant la fonction `run()`. La fonction principale à modifier est `handle_char()`.

Pour simplifier, on ne traite *en entrée* que des nombres formés d'un seul chiffre.

## 2.1 Ordre préfixé, ordre infixé, ordre postfixé

Nous avons l'habitude d'écrire les expressions arithmétiques en ordre infixé. Il existe aussi deux autres types de représentation des expressions arithmétiques qui ont l'avantage de ne pas avoir besoin de parenthèses. Ce sont les ordres préfixé et postfixé. L'ordre préfixé précise les opérateurs avant les arguments alors que c'est l'inverse pour l'ordre postfixé. Par exemple `((2+3)*6)` s'écrit `* + 2 3 6` en préfixé et `2 3 + 6 *` en postfixé.

## 2.2 Pile

Implémenter une structure de données de type *pile* : les opérations principales seront `push` et `pop`, qui permettent de rajouter et respectivement d'enlever un élément en tête de la pile d'évaluation . Il faudra aussi des opérations pour initialiser la pile, tester si elle est vide, etc.

La calculatrice fonctionne par effet sur une référence globale `global_stack_p` vers une pile d'évaluation.

## 2.3 Réalisation d'un évaluateur d'expression d'ordre postfixé

Écrire un programme qui calcule la valeur d'une expression postfixée.

Le programme est constitué d'une boucle qui lit une entrée[1] (chaîne de caractère) par itération et lui applique le traitement adéquat sur la pile globale d'évaluation.

Si c'est un opérateur `+`, `-`, `*`, `/`, `=`, il calcule à partir de la pile. Sinon, si c'est une valeur numérique, il la stocke dans une pile. L'opérateur `=` affiche la valeur du sommet de pile. Rajouter un symbole spécial ! qui a comme effet de vider la pile, et un symbole `.` qui termine le programme.

**Exemple.** On veut calculer : `2 3 + 6 *`. On rentre 2, qui est stocké dans la pile. Puis 3, qui va aussi dans la pile.`+` est un opérateur qui prend les deux éléments du dessus de la pile, calcule leur somme et la stocke dans la pile, i.e. 5. Puis, 6 est rentré et stocké dans la pile. `*` multiplie les 2 valeurs du dessus de la pile et stocke le résultat au sommet de la pile.

Essayer le calculateur avec les expressions suivantes :

```
4 5 8 3 * + - =
7 4 + 5 - 3 5 + 2 1 * - + =
1 2 3 4 5 6 7 8 1 1 +  + - * * + - + / =
```

## 2.4 Quelques extensions

### 2.4.1 Corrections

Permettre à l'utilisateur d'avorter l'évaluation d'une ligne ou d'effacer le dernier caractère rentré (touches `AC` et `C` sur la plupart des calculatrices).

### 2.4.2 Erreurs

Des message d'erreur explicites doivent être affichés. Proposer un système de "récupération", pour pouvoir reprendre le calcul même en cas d'erreur.

### 2.4.3 Terminaison

Demander confirmation à l'utilisateur avant de quitter le programme.

## 2.5 Affichage

Raffiner les fonctions de manipulation de la pile globale d'évaluation poru afficher les états successifs de la pile. Attention, vous ne devez rien changer au programme principal !

## 2.6 Compilation

Créer un fichier executable a partir de votre code OCaml, en utilisant la commande

```
ocamlc -o <fichier_executable> <fichier_source.ml>
```

[1]Pour la partie interface, se reporter au chapitre 16, "The core library", du manuel OCaml.

Pour des informations complémentaires sur la compilation, vous pouvez consulter le chapitre 7 “Batch compilation” du manuel en ligne.

# 3 Ordonnanceur de tâches (si vous avez du temps)

On se propose ici de simuler un petit ordonnanceur de tâches. Un tâche est juste une fonction. Pour que les choses soient intéressantes, il vaut mieux qu'elle ait un effet de bord permettant de différencier les appels successifs. Par exemple :

```
let f = let c = ref 0 in function () ->
      (print_string "\t f: "; print_int !c; print_newline (); c := !c + 1);;
```

## 3.1 Ordonnanceur simple

Un ordonnanceur est un programme qui agit sur une liste de tâches, maintenue dans une référence globale `global_list_p`. Écrire les fonctions qui permettent d'initialiser cette liste (`init_fun ()`), d'ajouter une fonction (`add_fun f`), d'enlever une fonction (`remove_fun f`), de sélectionner une fonction (`select_fun ()`).

À chaque itération, l'ordonnanceur choisit une fonction à exécuter, par exemple en parcourant la liste courante, et lance l'exécution :

```
while true do (select_fun ()) () done
```

## 3.2 Un peu plus compliqué

On veut maintenant que la fonction ordonnancée puisse agir sur l'ordonnanceur.

- Une fonction peut demander à être retirée de l'ordonnanceur en levant l'exception `Exit`.
- Elle peut abandonner son quantum d'exécution en levant l'exception `Yield`.
- Elle peut forcer l'exécution d'une autre fonction en la nommant explicitement : `Select f`.

## 3.3 Encore plus compliqué

Chaque fonction de la liste est maintenant équipée d'une priorité : un réel entre 0 et 100. Les fonctions sont choisies en fonction de leurs priorités relatives. Une fonction peut modifier sa priorité en levant une exception : `Priority 50`. Faire en sorte de ne pas avoir à recalculer à chaque fois la somme des priorités !

Ajouter aussi d'autres stratégies d'ordonnancement, par exemple *aléatoire*, parcours en sautant un élément sur deux, etc. Faire en sorte qu'une fonction puisse dynamiquement modifier la stratégie d'ordonnanement par exemple en levant une exception à laquelle la fonction de choix est attachée.