Programmation fonctionnelle

TP 3

Giron David thor@epitech.net

Résumé: Ce 3ème TP de programmation fonctionnelle vous familiarisera avec l'écriture de programmes avec OCaml à travers la préparation au projet `Bistromathique (Reloaded !)`*. Maintenant que vous connaissez la syntaxe et les bases, il est temps de passer aux choses sérieuses.*

Table des matières

Chapitre I

Premières compilations

Laissons derrière nous l'interprète OCaml et passons en mode compilé...

I.1 Exercice 1

- Lisez les (courts) mans de `ocamlc` et de `ocamlopt`.

I.2 Exercice 2

- Écrivez un Makefile pour compiler du code OCaml.
- La règle par défaut compilera vos sources vers du code natif.
- Une règle de votre choix compilera vos sources vers du bytecode.
- Votre `make` devra bien sûr inclure les règles habituelles `clean`, `fclean` et `re`
- Ajoutez toutes les règles qui vous semblent pertinentes.
- Prenez le temps de faire votre Makefile proprement et une bonne fois pour toute. Souvenez-vous : Un bon développeur est un développeur fainéant.

Si votre expérience des Makefiles se résume à copier/coller celui que vous avez récupéré en tech1, c'est l'occasion ou jamais de rattraper votre petit retard. Ce tutoriel, téléchargeable en PDF si vous êtes connecté sur le site, est très complet mais doit être lu en entier : http://bibliotech.epitech.eu/?page=book&id=158.
Pour aller plus loin avec les Makefile, nous vous conseillons le livre "Managing Projects with make" : http://bibliotech.epitech.eu/?page=book&id=76.

Chapitre II

Arguments d'un programme

Passer des arguments à un programme est bien évidemment possible en OCaml. La lecture de la documentation du module de la bibliothèque standard `Sys` nous apprend ceci :

```
1     val argv : string array
2          The command line arguments given to the process. The first
3          element is the command name used to invoke the
4          program. The following elements are the command-line
5          arguments given to the program.
```

`Sys.argv` est donc une valeur de type `string array`. Vous ne connaissez pas encore le type `'a array` que nous couvrirons en détails lors du prochain cours, mais un rapide coup d'oeil à la documentation permet de découvrir la fonction

`val to_list : 'a array -> 'a list`

du module `Array` qui repondra à tous nos besoins dans notre contexte.

II.1 Exercice 3

- Écrivez un programme qui affiche ses arguments sur la sortie standard.
- Vous devez utiliser les fonctions `Array.to_list`, `List.iter` et `print_endline` dans votre programme.

Chapitre III

Parser les arguments d'un programme

La bibliothèque standard d'OCaml propose un module pour parser les arguments passés à un programme. Ce module est `Arg` et rapellera la fonction `getopt` de la `libc` à ceux qui la connaissent.

III.1 Exercice 4

L'utilisation de ce module peut conduire à l'utilisation de références qui seront couvertes en détails dans le prochain cours. En attendant, voici un petit exemple de code qui devrait vous en apprendre suffisement sur les références pour utiliser le module `Arg` :

```
1  # let x = ref 0;; (* notez le "ref" *)
2  val x : int ref = {contents = 0}
3  # !x;; (* notez le '!' *)
4  - : int = 0
5  # x := 1;; (* notez le ":=" *)
6  - : unit = ()
7  # !x;; (* notez le '!' *)
8  - : int = 1
```

- Lisez la (courte) documentation du module `Arg` à votre disposition à l'adresse http://www.janestreet.com/ocaml/janestreet-ocamldocs/caml/Arg.html

III.2 Exercice 5

- Écrivez le parseur d'options de votre projet `Bistro (Reloaded!)` en utilisant le module `Arg`.
- Si l'argument n'existe pas, vous devez afficher un message d'erreur.
- Dans le cas où l'argument `-obase` est passé avec une mauvaise valeur (i.e. différente de 2, 8 10 et 16), vous devez afficher un message d'erreur.
- Dans le cas où votre programme est appelé sans argument, vous devez afficher un `usage`.

Contrairement à ce qu'on pourrait croire, `Arg.parse` parse bien la ligne de commande alors que `Arg.parse_argv` parse un tableau passé en paramètre comme si ce tableau était celui de la ligne de commande. La fonction qui vous intéresse pour cet exercice est bien `Arg.parse` et non `Arg.parse_argv`. Ainsi, vous n'aurez pas à vous préoccuper de `Sys.argv`.

Chapitre IV

Tordre le cou à Eval Expr

Vous avez deux approches possibles pour cette partie du TP : soit la faire sérieusement en réfléchissant, soit recopier l'exemple du cours et rentrer chez vous jouer à World Of Warcraft © ou toute autre activité qui vous rendra plus beau. À vous de voir.

`Eval Expr` est un problème canonique du débutant en programmation. À ce titre, il est donné aux etudiants sortant de piscine C en première année afin de les préparer au projet `Bistromathique`. Dans le même esprit, vous allez maintenant écrire un `Eval Expr` pour vous introduire au projet `Bistromathique (Reloaded !)`. Cependant, on ne s'intéressera ici qu'à sa partie interprétation et non au parsing des expressions.

`Eval Expr` n'est en réalité qu'un cas particulier d'un interprète : l'interprète des expressions arithmétique simples. Un interprète se résume généralement à un simple parcours en profondeur d'un arbre construit par le front-end de l'interprète. Un tel arbre est appelé un "arbre de syntaxe abstraite" dont le rôle est de représenter l'expression à interpréter sous une forme adaptée. On peut donc distinguer deux problématiques liées qui définissent un interprète :

- Une forme adaptée de l'arbre à parcourir.
- Des actions à effectuer lors du parcours pour les nœuds et les feuilles de l'arbre.

IV.1 Exercice 6

Commençons par déterminer une forme adéquate pour représenter une expression arithmétique sous forme d'arbre. Le type de données adéquat en OCaml est le variant (ou type algébrique ou encore type somme, comme vous préférez). N'oubliez pas que votre variant peut être récursif (et doit l'être dans notre exercice).

- Définissez un type `expression` pouvant représenter des sommes, des différences, des produits, des quotients et des valeurs entières. Le choix des constructeurs est à votre discrétion.

Ce sont les règles de priorité des operations qui vont determiner la forme de votre arbre. Par exemple, l'expression `1+(2*3)` est très différente de l'expression `(1+2)*3` bien que l'ordre des opérandes et des opérateurs soit le même.

- Entraînez-vous à représenter les valeurs suivantes avec votre type `expression` en suivant les règles de priorité que vous connaissez :
  - 1+1
  - 1+1+1
  - (1+1)+1
  - 1+(1+1)
  - 1+1+1+1
  - 1+2x3
  - (1+2)x3
  - 1-2x3/4
  - (1-2)x3/4
- Comparez vos valeurs de type `expression` avec votre voisin...

Plusieurs parmi vous n'aurons peut-être pas représenté leurs expressions de la même façon que leur voisin bien que l'évaluation des deux arbres donne le même résultat !

L'ambiguïté entre ces arbres met en lumière une foule de problématiques de théorie des langages passionnantes mais sortant largement du cadre d'`Epitech`. Si vous êtes curieux, parcourez donc le net avec des mots clés comme "automates", "grammaires", "déterminisme", "Chompsky". Les liens feront le reste...

Dans le cas qui nous intéresse, la problématique est l'associativité des opérateurs. L'expression "1+1+1" doit-elle être représentée "(1+1)+1" ou "1+(1+1)" ? La réponse est bien sûr la première car vous savez empiriquement que l'opérateur "+" est associatif à gauche.

- Assurez-vous que vos valeurs de type `expression` respectent les règles d'associativité.
- Connaissez-vous au moins un opérateur associatif à droite ? Cherchez bien...

IV.2 Exercice 7

Maintenant que vous avez quelques arbres prêts à être interprétés, nous pouvons écrire l'interprète en question.

- Écrivez une fonction récursive `eval_expr` de type `expression -> int` qui calcule le résultat de l'expression passée en paramètre et qui l'affiche sur la sortie standard.
- Votre fonction devra faire un parcours en profondeur à gauche de l'arbre représentant l'expression.
- Bonus frappe : Pourquoi un parcours en profondeur à droite n'est-il pas envisageable sans modification de l'arbre ?

IV.3 Exercice 8

Pour aller plus loin :

- Écrivez une fonction récursive `compile_expr` de type `expression -> string` qui renvoie une chaîne de caractères représentant le code dans le langage de votre choix qui calcule cette expression.
- Écrivez cette chaîne de caractères dans un fichier.
- Compilez le fichier avec le compilateur approprié.
- Constatez que vous avez ecrit un programme qui génère des programmes.

Bon courage !

Chapitre V

Conclusion

Nous espérons que vous avez apprécié ce sujet autant que nous avons apprécié le rédiger pour vous.

Vos avis sont très importants pour nous et nous permettent chaque jour d'améliorer nos contenus. C'est pourquoi nous comptons beaucoup sur vous pour nous apporter vos retours.

Si vous trouvez que certains points du sujet sont obscurs, pas assez bien expliqués ou tout simplement contiennent des fautes d'orthographe, signalez-le nous. Pour cela, il vous suffit de nous envoyer un mail à l'adresse koala@epitech.eu.

Pour aller plus loin dans votre apprentissage de la programmation fonctionnelle, nous vous conseillons le livre "Développement d'applications avec Objective CAML" : http://bibliotech.epitech.eu/?page=book&id=156 ou sa version en anglais dont le PDF est disponible : http://bibliotech.epitech.eu/?page=book&id=155.

Il existe bien sûr d'autres livres sur la programmation fonctionnelle : http://bibliotech.epitech.eu/?page=search&categ=15. N'hésitez pas à vous renseigner auprès des koalas, ils seront ravis de vous conseiller. Certains livres sont disponibles en version PDF si vous êtes connecté sur le site.