Projet 2 L3IF : interpréter, compiler

Rendu 2 : pour le 22 mars 2018 à 23h59

par mail à `henning.basold@ens-lyon.fr`,
`daniel.hirschkoff@ens-lyon.fr`,
`bertrand.simon@ens-lyon.fr`

(deux autres rendus suivront, dans le prolongement de celui-ci)

1 Le langage de départ : fouine

On s'intéresse à fouine, un sous-ensemble de Caml qui est décrit en `http://www.ens-lyon.fr/DI/?p=5451`.

Le but est d'écrire un interprète, mais attention, pas au sens d'un programme interactif qui propose, comme OCaml, de saisir des expressions au clavier et de les évaluer dans la foulée.

L'interprète prend en entrée un fichier Caml, exécute le code qui s'y trouve, et affiche ce qu'on lui demande d'afficher.

2 Débutants

Vous devez coder l'interprète pour le sous-ensemble de fouine comprenant :

1. les expressions arithmétiques ;
2. les `let...in` (avec toutes les écritures acceptées dans la spécification de fouine — avec les `;;`, etc.).

   On vous conseille de commencer par avoir une version qui marche sans accepter les écritures alternatives pour les `let.. in`.
3. le `if then else`, tel que spécifié dans fouine ;
4. la "fonction" `prInt`.

Vous devez aussi proposer l'option `-debug` (cf. partie 5).

Si vous êtes débutants, vous pouvez passer directement à la partie 5.

3 Intermédiaires

Comme toujours, le menu c'est "débutants" + ce qui suit.

1. les fonctions (et donc les clôtures : la première chose à faire est de relire les transparents du cours à ce propos) ; une difficulté du côté de l'analyse lexicale et syntaxique est le traitement de l'application, en lien notamment avec le moins unaire (`3 -1` c'est une expression arithmétique, pas l'application de `3` à `-1`) ;
2. les fonctions récursives.

Si vous êtes intermédiaires, vous pouvez passer directement à la partie 5.

4 Avancés

Comme toujours, le menu c'est "intermédiaires" + ce qui suit.

1. Les aspects impératifs.

   Constructions Caml à traiter : `:=`, `!`, `;`, `ref`, `()`.

   On autorisera les références sur des entiers, sur des fonctions, et sur des références. Un calcul pourra renvoyer une réference, ou alors `()` (comme par exemple dans `let _ = a := a+1` — à noter que `prInt` continue à renvoyer un entier). On se contentera d'utilisations simples de `ref`, qui ne sera pas traitée comme une "vraie fonction"[1].

   Important : vous n'avez pas le droit d'utiliser les références de Caml pour implémenter directement les références de fouine. Cela signifie en particulier qu'il vous faudra implémenter une structure de données servant de représentation de la mémoire.

   Il vous est suggéré de ne traiter dans un premier temps que les références sur des entiers, pour ensuite passer à la généralisation.

   Vous pouvez décider de traduire une séquence (comportant un `;`) en un `let...in`.

2. Les couples.

   La virgule, la "déconstruction" d'un couple de la forme `let (x,y) = c in...`. Vous pouvez aussi autoriser `match c with | (x,y) -> ...`, mais ça n'est pas obligatoire.

Bonus. *Si tout le reste est traité, et de manière propre,* vous pouvez ajouter les types somme ; cette partie est facultative.

- Soit vous ajoutez juste les listes, avec `[], ::`, et le filtrage (avec `match...with` et/ou avec `function`). L'interprète pourra engendrer un `Warning` si un filtrage ne comporte qu'un cas sur deux.
- Soit vous ajoutez la possibilité de définir et manipuler des types somme. À noter que comme on ne fait pas de vérification de types, la définition d'un nouveau type somme revient à donner une liste finie de constructeurs (par exemple, `type tree = Leaf | Node`, l'idée étant que comme pour les autres constructions, une erreur à l'exécution peut survenir si on fait n'importe quoi avec les types).

5 Spécification : exécutable, options, etc.

Exécutable et options. Le suffixe pour les fichiers en fouine est `.ml` . L'exécutable s'appelle `fouine`, on l'appelle de la manière suivante : `./fouine toto.ml` (pas de `fouine < toto.ml`).

Sans options, le programme exécute l'interprète, et affiche ce qu'on lui dit d'afficher (à l'aide de `prInt`).

`-debug` Cette option aura pour effet d'afficher le programme écrit en entrée (cf. paragraphe "Affichage" dans la description du langage fouine), en plus de l'affichage qui est engendré si on n'appelle pas `-debug`.

Vous pouvez bien sûr afficher davantage d'informations lorsque l'utilisateur choisit cette option, informations qui permettront de suivre l'exécution du programme.

Un exemple est disponible sur la page www du cours pour vous aider à gérer l'exécutable et les options.

[1] Alors qu'en Caml :

```
# ref;;
- :  'a -> 'a ref = <fun>
```

Tests.

- Un certain nombre de fichiers de tests vous seront/sont fournis à partir de la page www du cours. Vos programmes doivent tourner correctement sur ces tests.
- Fournissez un répertoire avec des programmes de test que vous avez soumis à `fouine` (inutile d'y inclure ceux que nous vous fournissons). Veillez à ce que les tests couvrent l'ensemble des aspects que vous traitez (lex-yacc, exécution des divers composants du langage fouine).
- Indiquez s'il y a des tests qui échouent, en raison de bugs pas encore résolus.

Le rendu. L'approche est la même que pour le rendu précédent, en ce qui concerne la propreté du rendu, les explications, etc.

Il est *important* que vous vous répartissiez clairement le travail, afin que l'on puisse déterminer qui a fait quoi. Et il est indispensable d'indiquer cette répartition dans le README que vous incluez dans le rendu.

Veillez à la propreté de votre rendu : il y a le temps de peaufiner les détails, nettoyer/commenter le code, vérifier que tout marche, et envoyer quelque chose de "fini".

Planification. Ce rendu met un accent assez fort sur l'organisation du travail : bien plus que pour le rendu 1, il sera important de progresser de semaine en semaine pour espérer aboutir à ce qui est attendu au bout de trois semaines.

Voici, à titre indicatif, des suggestions d'échéanciers :

- pour les binômes Débutants
  1. tout (parser, interprète, tests) pour le langage sans `let.. in` ;
  2. ajout de l'environnement : `let.. in`, variable ;
  3. ajout du reste, peaufinage.
- pour les Intermédiaires
  1. expressions avec `let.. in` ;
  2. fonctions ;
  3. fin des fonctions (ça n'est pas simple...) et le reste (dont fonctions récursives).
- pour les Avancés
  1. Tout jusqu'aux fonctions ;
  2. tout sauf les aspects impératifs ;
  3. aspects impératifs.

À chaque fois, l'idée est que chaque semaine correspond à un rendu que vous ne nous envoyez pas : tout marche, il y a un README (certes minimal, au début), et des fichiers de tests. Si vous pensez être pris par le temps, vous pouvez "viser" les points 1 et 2 uniquement (mais cela vous pénalisera, bien sûr).