Second demi-projet : interpréter, compiler

Rendu 2 : pour le 11 avril à 23h59

par mail à `aurore.alcolei@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. On gardera la même syntaxe.

1.1 Syntaxe du langage

Cœur du langage fouine.

- expressions arithmétiques : constantes entières (y compris négatives), somme, soustraction, multiplication ;
- fonctions : on peut délarer une fonction avec `let f x z = ..` ou avec `let f = fun x -> fun z -> ..` (ou un mélange) ; on peut appliquer les fonctions comme dans `f s t` et on peut utiliser une fonction directement sans la nommer `(fun x -> x) a`.
- construction `let .. in`

  NB : vous pouvez, si vous le souhaitez, autoriser les séquences de `let`, sans le `in`, comme il est possible de le faire en Caml (p.ex., `let x = 2  let y = x*3`). Mais cela n'est pas demandé explicitement.
- construction `let rec` (en deux mots), pour définir des fonctions récursives (pas de message d'erreur si la fonction n'est pas récursive). On ne s'intéressera qu'à des définitions récursives de fonctions.
- parenthèses ouvrantes et fermantes, sauts à la ligne autorisés ;
- `if then else`, avec des tests entre entiers ;

  opérateurs autorisés : `> >= < <= = <>` (vous pouvez ajouter `not && ||`, mais ça n'est pas imposé)
- on met un `;;` pour terminer une expression. Vous pouvez si vous le souhaitez accepter la version où le `;;` n'est pas mis.
- un "programme" à exécuter est une simple expression, qui peut comporter des `let .. in` (mais n'est pas, par exemple, juste `let a = 3`).
- vous pouvez ajouter `begin..end` (ça n'est pas très difficile), ainsi que `_` (il faut faire un peu plus attention).

Priorités etc. Là aussi, le comportement de Caml sert de référence. La principale difficulté est le traitement des applications lorsqu'une fonction est appliquée successivement à plusieurs arguments. Même chose lorsque l'on définit une fonction sans utiliser `fun .. ->`

Vous n'avez en particulier pas le droit d'imposer à l'utilisateur d'ajouter davantage de parenthèses que ce qui est demandé en Caml.

Voici quelques exemples illustratifs :

```
# let add x y = x+y;;
val add : int -> int -> int = <fun>
# 3 + add 2 3;;
- : int = 8

# let k = 3 + (fun x -> x + 1) 5;;
val k : int = 9
# let k = 3 + fun x -> x + 1 5;;
Characters 12-28:
  let k = 3 + fun x -> x + 1 5;;
              ^^^^^^^^^^^^^^^
Error: This expression should not be a function, the expected type is int
```

La fonction `prInt`. On ajoute une constuction spéciale au langage, pour pouvoir faire de l'affichage et ainsi voir ce que font les programmes. Cette construction s'appelle `prInt`, c'est une fonction qui affiche son argument entier, et renvoie pour valeur ce même argument.

Voici un petit exemple :

```
# 2 + prInt (1+2);;
3
- : int = 5
```

En Caml (mais pas en fouine), on peut *définir* `prInt` comme suit :

```
# let prInt x =
  print_int x;print_newline(); x;;
  val prInt : int -> int = <fun>
```

Affichage. Vous devez programmer une fonction qui affiche un programme fouine. Afficher le programme saisi en entrée ne donne pas nécessairement le même programme, car il se peut que des transformations élémentaires soient appliquées au passage (par exemple, `let f x =..` remplacé par `let f = fun x -> ..`). Il faudra toutefois que le programme affiché soit accepté par Caml, à la fonction `prInt` près.

1.2 Exécution des programmes

Plutôt que de décrire formellement la sémantique opérationnelle de fouine, on adopte l'approche *"lancez le programme avec Caml, cela vous donnera le comportement attendu"*. Vous pouvez interagir avec les encadrants si vous avez des doutes sur un point précis.

Typiquement, de petits programmes comme `let a = prInt 5 in a + prInt 7` ou `let f x y = x*y in f (prInt 3) (2+prInt 2)` vous permettront de vérifier que vous implémentez les choses comme il faut.

Les erreurs que ne connaît pas Caml. En Caml, un programme est typé, ce qui permet d'éviter d'exécuter des programmes absurdes comme `3 + (fun x -> x+1)`. Pas de typage en fouine, du coup certains programmes exploseront lors de l'exécution.

Réfléchissez à la gestion de ces erreurs (appelées *dynamiques*), et proposez une solution plus ou moins satisfaisante. Cela peut aller de messages d'erreur un peu explicatifs dans certains cas (quand une addition fait intervenir une fonction, par exemple) à des choses plus sophistiquées, où l'on indique d'où vient l'erreur dans le code source.

2 Un interprète

2.1 L'interprète de base, avec des environnements et des clôtures

Développez votre interprète en plusieurs étapes, surtout pour les **D** (vous pouvez mettre des `failwith` dans votre code pour gérer temporairement les cas qui ne sont pas encore traités) :

1. expressions arithmétiques, `if then else` et `prInt` **D I A**
2. `let .. in` **D I A**
3. fonctions pas récursives, avec les clôtures **D I A**
4. fonctions récursives **I A**

Remarque pour les I. Il vous est demandé de traiter l'une des deux extensions décrites dans ce qui suit.

2.2 Extension 1 : exceptions I A

Syntaxe.

- il y a une seule exception, qui est notée `E`, et prend un entier en argument ;
- il y a, comme en Caml, les constructions `raise` et `try .. with` ;
- exemples : `raise 17` `let k = try f x with E x -> x+1`

  NB : on omettra le "|" facultatif dans le `with E x -> ...`

Implémentation.

1. Étendez lex et yacc pour traiter les programmes avec exceptions.
2. Étendez l'interprète pour prendre en compte les exceptions. Comprenez au passage comment sont gérés les environnements lors que l'on "saute" lors d'un `raise`.

2.3 Extension 2 : références et aspects impératifs I A

Syntaxe.

- On pourra utiliser `ref` pour créer, comme en Caml, des références. Mais on ne pourra créer *que des références à des entiers* (pas à des fonctions).
- On a, comme en Caml, `:=`, `!` et `;`

  Une opération `:=` renvoie un entier (celui qui a été affecté à la référence) — il s'agit ici d'une petite différence vis-à-vis de Caml. On évitera cependant d'écrire des programmes tirant parti de cette particularité, du style `2+(r := 4)`, qui sont rejetés par Caml.

Implémentation.

1. Étendez lex et yacc pour traiter les programmes avec références.
2. Étendez l'interprète pour prendre en compte les références.

Vous pouvez implémenter les références ... à l'aide de références, ce qui est sans doute le plus simple. Vous pouvez aussi "mimer" une mémoire, où une référence sera représentée par un entier correspondant à une case dans un tableau.

Extensions de l'extension. A Les deux extensions suivantes sont facultatives.

Traitez le cas des références dans le cas général (on peut stocker des fonctions dans les références).

Ajoutez des tableaux d'entiers, pour pouvoir écrire des programmes qui "font quelque chose", à part additionner et multiplier des entiers. Les tableaux seront manipulés à l'aide des primitives suivantes :

- `aMake` prend un argument un entier k et renvoie un tableau de taille k, initialisé à 0 partout ;
- comme en Caml, on utilise `t.(1)` pour accéder à la deuxième case du tableau `t`, et on utilise `<-` pour modifier une case du tableau.

On peut en Caml définir `amake` de la façon suivante :

```
# let aMake = fun k -> Array.make k 0;;
val aMake : int -> int array = <fun>
```

3 Un compilateur vers une machine à pile

3.1 La machine SECD A

Programmer une compilation vers la machine SECD telle que vue en cours, pour le langage de départ (fonctions récursives exclues, `prInt` inclus).

Ajouter dans un second temps les fonctions récursives.

Documentation en ligne, améliorations possibles. Vous trouverez ici les transparents d'un cours de Xavier Leroy d'où vient ce qui vous a été raconté en cours (qui est disponible ici).

Une première amélioration consiste à passer en *indices de De Bruijn* pour la représentation des lieurs : plutôt que d'avoir `let x = 3 in let y = 4 in x+y`, on a quelque chose comme `let 3 in let 4 in 1+0` (pas dans le source, bien sûr, mais dans la représentation interne du code). Ceci permet d'avoir des ACCESS(i) plutôt que des ACCESS(x) dans la machine.

Si vous êtes motivés, vous pouvez aller regarder la machine Zinc, afin d'améliorer les performances de la SECD (mieux gérer les fonctions à plusieurs variables). Idéalement, vous comparerez les performances des deux machines sur une batterie d'exemples. Des liens à propos de la Zinc: ici et ici.

Vous pouvez aussi ajouter les extensions décrites plus haut (références, exceptions).

4 L'alternative, ou le bonus (binômes fort avancés) FA

4.1 De quoi il s'agit

L'article disponible ici décrit une manière originale d'exécuter des programmes fonctionnels simples[1].

Il vous est proposé de :

1. lire et comprendre l'article, ou tout du moins les parties nécessaires au traitement de ce rendu.
2. implémenter le langage décrit à la partie 1 de cet énoncé en suivant l'approche de l'article
3. ajouter le type `unit`
4. ajouter les références, comme décrit à la partie 2.3 de cet énoncé
5. discuter de l'ajout des exceptions (voir la partie 2.2 de cet énoncé)

[1] Le code qui va avec se trouve ici.

4.2 Protocole

Vous pouvez, si vous êtes fort avancés[2], décider d'opter pour ce qui suit :

- vous faites d'une part l'interprète pour tout le langage fouine (avec références et exceptions) ;
  (ou alors juste la compilation vers la machine plutôt que l'interprète, si vous préférez)
- d'autre part, vous traitez la partie 4.1 (au long des rendus 2,3,4 — les échéances seront à discuter avec D. Hirschkoff).

À noter qu'il s'agit là d'un travail bien moins guidé que dans sa version "standard", il faudra prendre des initiatives et réfléchir pas mal de votre côté. Vous pouvez bien sûr poser vos questions à D. Hirschkoff s'il y a des points que vous trouvez obscurs dans l'article.

La première chose à faire, si vous êtes intéressés, est d'en faire part à D. Hirschkoff par mail.

5 Spécification : exécutable, options, etc.

Le suffixe pour les fichiers en fouine est `.ml` . L'exécutable s'appelle `fouine` .

Options.

Sans options, le programme est exécuté avec l'interprète.

`-debug` Cette option aura pour effet d'afficher le programme écrit en entrée (cf. paragraphe "Affichage" à la fin de la section 1.1). Vous pouvez bien sûr afficher davantage d'informations lorsque l'utilisateur choisit cette option, qui premettront de suivre l'exécution du programme.

`-machine` compile le programme vers la machine, puis l'exécute (pour les **D**, cette option correspondra à l'exécution "hybride" interprète/machine, à partir du rendu 3).

`-interm toto.code` affiche le programme compilé, sans l'exécuter.

`-NbE` si vous traitez la partie 4

Des tests. Fournissez un répertoire avec des programmes de test que vous avez soumis à `fouine`. 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. Vous pouvez vous reporter au rendu 1 et au DM : l'esprit est le même, en ce qui concerne la propreté du rendu, les explications, etc.

Au sujet de la 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.

À titre indicatif, pour les binômes **D**ébutants, une suggestion d'échéancier est la suivante :

1. tout (parser, interprète, tests) pour le langage sans variables et sans fonctions ;
2. ajout de l'environnement : `let.. in`, variable ;
3. ajout des fonctions et des clôtures.

À chaque fois, l'idée est que chaque semaine correspond à un rendu : tout marche, il y a un README (certes minimal, au début), et des fichiers de tests.

[2] Qualificatif qu'il est possible de négocier...