Second demi-projet : interpréter, compiler

Rendu 3 : pour le 2 mai à 23h59

par mail à `aurore.alcolei@ens-lyon.fr`,
`daniel.hirschkoff@ens-lyon.fr`,
`bertrand.simon@ens-lyon.fr`

(un autre rendu suivra, dans le prolongement de celui-ci)

1 Préambule

*Que vous soyez **D**, **I** ou **A**, ce sujet est écrit en faisant l'hypothèse que tout a été traité (plus ou moins bien...) lors du rendu 2.*

Si tel n'est pas le cas, envoyez rapidement *un mail à* `daniel.hirschkoff@ens-lyon.fr` *pour indiquer ce qui manque dans votre rendu 2 par rapport au menu escompté.*

Le but étant de ne pas trop "charger la barque", et de maintenir une exigence raisonnable.

À noter que l'on vous demande de corriger le code vis-à-vis de la typo qui était dans le sujet du rendu 2, sur le `raise` *(cf. note de bas de page).*

2 L'interprète D I

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

Les **D** doivent maintenant faire ce qu'on fait les autres lors du précédent rendu :

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

Remarque pour les I. Il vous est demandé de traiter les deux extensions décrites dans les deux parties qui suivent (2.2 et 2.3).

2.2 Extension 1 : exceptions I

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 (E 17)`[1] `let k = try f x with E x -> x+1`

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

[1] Il s'agit là d'une typo dans le sujet du rendu 2, où il y avait écrit `raise 17`.
Merci de modifier, le cas échéant, pour le rendu 3.

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 lorsque l'on "saute" lors d'un `raise`.

N'implémentez pas une exception en utilisant une exception de Caml. Enrichissez plutôt l'interprète avec un argument supplémentaire (une pile), qui servira à gérer les mécanismes des exceptions (il faudra comprendre ce que contient un élément de la pile, et quand est-ce qu'on empile/dépile).

2.3 Extension 2 : références et aspects impératifs I

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.

2.4 Extensions de l'extension.

Les deux extensions suivantes sont facultatives.

Traitez le cas des références dans le cas général : on peut stocker des fonctions, ou des références, dans les références. La mise à jour d'une référence peut renvoyer un entier arbitraire, ou alors, pour être plus fidèles à Caml, vous pouvez enrichir fouine avec le type `unit`.

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 Une machine à pile pour les expressions arithmétiques D I

Il vous est demandé de traiter l'option `-machine` et `-interm` du compilateur, de la manière suivante :

- vous prenez en entrée un programme fouine ;
- si celui-ci ne comporte que des expressions arithmétiques (sans `let in`, sans fonction), vous passez à la suite, sinon vous affichez un message d'erreur et tout s'arrête ;
- vous compilez le programme fouine vers un programme pouvant s'exécuter sur une machine à pile (comme vu en cours) ;
- vous exécutez le programme sur la machine à pile (que vous aurez donc implémentée).

À noter que la construction `prInt` doit également être traitée : on affiche l'entier, et on passe à la suite.

3.2 La machine SECD A

Terminez la machine SECD, en traitant références et exceptions (dans l'esprit de ce qui a été fait pour l'interprète). À noter que pour le traitement des exceptions, vous enrichirez la machine avec une composante supplémentaire, et nous appuierez pas sur le mécanisme d'exceptions de Caml.

Vous pouvez aussi vous intéresser à des raffinements de la machine vue en cours, comme l'indique le paragraphe suivant.

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.

4 Extensions du langage et transformations de programmes A

Le but de cette partie est d'implémenter les transformations de programmes vues lors de la dernière séance en amphi. L'approche est la suivante :

1. on récupère un programme fouine "étendu" grâce aux lexer&parser ;
2. on applique la transformation, de manière à obtenir un programme fouine sans extension ;

   on est alors capable d'afficher le programme résultant (même si on ne sait pas si on a vraiment envie de le contempler) ;
3. on interprète le programme ainsi obtenu avec l'interprète programmé au rendu 2 (si on dispose aussi de la machine, on peut compiler le programme ainsi obtenu, et l'exécuter).

4.1 Étendre fouine

Pour implémenter les transformations de programmes, il vous faudra étendre fouine avec les couples. Voici la syntaxe : `(e1,e2)` (éventuellement sans les parenthèses, si vous le sentez), et `let (x,y) = e in e'` pour déconstruire un couple.

Ajoutez cela à la syntaxe fouine, et à l'interprète du rendu 2 (vous pouvez aussi étendre à la partie "compilation", mais ça n'est pas explicitement demandé).

4.2 Exceptions

L'extension de fouine est décrite à la partie 2.2, la transformation vue en cours s'appuie sur le style "par continuations"[2]. Il s'agit d'implémenter la transformation de fouine+exceptions vers fouine.

Extension possible: ajoutez aussi `call/cc` dans le langage source (en regardant sur internet comment ça marche — `call/cc` signifie *"call with current continuation"*, donc justement).

4.3 Références et aspects impératifs

L'extension de fouine est décrite à la partie 2.3, la transformation vue en cours s'appuie sur le style "par passage d'état" (*state passing*). Il s'agit d'implémenter la transformation de fouine+impératif vers fouine.

Pour cette transformation, il vous faudra aussi avoir des primitives permettant d'accéder à la mémoire : `alloc`, `read`, et `write`. Afin d'en disposer dans le langage cible de la transformation, vous pouvez "tricher", et vous appuyer sur l'extension avec les tableaux décrite à la partie 2.4 (extension que vous devrez donc implémenter). Si vous voulez être davantage dans l'esprit, vous pouvez concevoir une implémentation purement fonctionnelle de ces trois primitives (à coup de paires, ou alors en étendant fouine avec des listes).

N'oubliez pas d'expliquer dans le README comment vous avez procédé (ceci vaut d'ailleurs pour un peu toutes les parties de tous les rendus).

4.4 Combiner les deux transformations

Implémentez une transformation de fouine+exceptions+impératif vers fouine. Il vous faudra pour ce faire réfléchir à comment définir cette transformation (et dire quelques mots là-dessus dans le README).

Autant que faire se peut, privilégiez l'élégance et la modularité dans le code : il serait bien d'éviter d'avoir 3 fonctions ne partageant rien pour implémenter les trois transformations demandées dans cette partie 4.

Améliorations possibles. Vos transformations pourraient traiter "astucieusement" la curryfication, en évitant d'appliquer systématiquement la transformation de base lorsqu'on a affaire à une définition du style `let f x y z = ....`

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.

Il affiche le résultat de l'évaluation, à savoir :

- soit un entier,

[2] On rappelle les cas de l'application et du let:

```
[e1 e2] = fun (k,kE) -> [e2](fun v -> [e1](fun f -> f v k, kE) , kE)
[let x = e1 in e2] = fun (k,kE) -> [e1](fun x -> [e2](k,kE) )
```

– soit une fonction (ou plutôt une clôture) : si vous avez exécuté la machine à pile, vous afficherez le code (de la machine) correspondant ; sinon, vous afficherez un programme Caml correspondant au résultat du calcul.

`-debug` Cette option aura pour effet d'afficher le programme écrit en entrée (cf. paragraphe "Affichage" à la fin de la section 1.1), 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, qui premettront de suivre l'exécution du programme.

`-machine` compile le programme vers la machine, puis l'exécute. Si vous avez codé plusieurs versions de la machine, l'option `-machine` prendra un argument, et vous décrirez cela dans le fichier README.

`-interm toto.code` engendre le programme compilé, dans le fichier `toto.code`, sans l'exécuter.

`-NbE` si vous traitez la partie 4 du rendu 2.

`-E` Applique la transformation pour éliminer les exceptions (section 4.2), puis exécute le programme ainsi obtenu à l'aide de l'interprète (rendu 2).

En combinant `-E` et `-debug`, on peut visualiser le programme obtenu après avoir appliqué la transformation (même chose pour `-R` et `-ER`).

`-R` Même chose, pour la transformation pour éliminer les aspects impératifs (section 4.3).

`-ER` La combinaison des deux transformations

Les quatre dernières options peuvent être combinées avec l'option `-machine` pour faire tourner le programme résultant à l'aide de la machine plutôt qu'avec l'interprète.

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.

Si vous traitez (même partiellement) la machine à pile, décrivez sommairement dans le README les instructions qui peuvent s'exécuter sur ladite machine.