Projet programmation, fouine, suite

Rendu 3 : pour le 1er avril à 23h59

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

0 Prélude pour tout le monde : traitement des déclarations avec ;;

La spécification de fouine sur la page www correspondante pour le rendu 2 n'était pas correcte, dans la mesure où elle autorisait

```
let a = 2 in
let b = 4;;
let c = 5 in
a*b*c
```

Si vous avez respecté la consigne, votre fouine accepte des programmes qui sont refusés par Caml.

Reprenez votre code de manière à refuser les écritures comme dans l'exemple ci-dessus, tout en respectant la nouvelle consigne, qui autorise

```
let a = 2;;
let b = 4;;
let c = 5 in
a*b*c
```

tout comme

```
let a = 2;;
let b = 4 in
let c = 5 in
a*b*c
```

ou

```
let a = 2;;
let b = 4;;
let c = 5;;
a*b*c
```

1 Débutants

Si cela n'était pas traité dans votre rendu 2, prenez en compte les options de debug du sujet du rendu 2 (cela devrait être une formalité : ne perdez pas trop de temps dessus).

Autres points relativement mineurs à prendre en compte :

- La fonction d'évaluation (appelée sans doute `eval`) et ce qui va avec (définition des valeurs et des environnements, typiquement) ne devraient pas être dans le même fichier que la définition des expressions (fichier `expr.ml`, sans doute) : créez un nouveau fichier pour cela, et recollez les bouts pour que tout marche bien.

- Si la définition du type `expr` ressemble à cela

  ```
  type expr =
    Pl of expr*expr
  | Mi of expr*expr
  | Mu of expr*expr
  | Di of expr*expr
  ...
  ```

  alors la fonction d'évaluation contient sans doute du code qui ressemble à cela (je caricature peut-être un peu) :

  ```
  | Pl(e1,e2) -> let v1 = eval env e1 in let v2 = eval env e2 in let k1 = val2int v1 in let k2 = val2int v2 in Cst (k1+k2)
  | Mi(e1,e2) -> let v1 = eval env e1 in let v2 = eval env e2 in let k1 = val2int v1 in let k2 = val2int v2 in Cst (k1-k2)
  | Mu(e1,e2) -> let v1 = eval env e1 in let v2 = eval env e2 in let k1 = val2int v1 in let k2 = val2int v2 in Cst (k1*k2)
  | Di(e1,e2) -> let v1 = eval env e1 in let v2 = eval env e2 in let k1 = val2int v1 in let k2 = val2int v2 in Cst (k1/k2)
  ```

Avoir 4 lignes identiques à un caractère près est désolant.

Reprenez le type `expr` pour factoriser, par exemple ainsi :

```
type arithop = Pl | Mi | Mu | Di

let arop2fun o = match o with
| Pl -> fun x y -> x+y
| ...

type expr = ArOp of arithop*expr*expr | ...

let rec eval = ....
| ArOp(op,e1,e2) -> let v1 = eval env e1 in let v2 = eval env e2 in \dots
```

Même idée pour les opérateurs booléens.

Traitez le menu Intermédiaires du rendu 2 : fonctions, fonctions récursives.

Pour ce faire, commencez par relire les transparents du cours. Étendez ensuite le type des expressions, et le type des valeurs (afin de prendre en compte la notion de clôture). Il est conseillé d'échanger avec les encadrant.e.s avant d'attaquer l'analyse lexicale pour les applications.

Un petit exercice utile pourrait être de s'astreindre à faire une grammaire pour les expressions constituées uniquement d'identifiants et de parenthèses, afin de reconnaître correctement `f u (g x) y`, par exemple. Lorsque ça marche, on peut ajouter l'opérateur de somme, et ainsi de suite en enrichissant progressivement le langage.

2 Intermédiaires

Si cela n'était pas le cas dans votre rendu 2, traitez `prInt` comme une "vraie fonction" : elle peut en particulier être passée à une autre fonction.

Traitez la partie "Avancés" du rendu 2 : traits impératifs, couples/tuples, listes.

Vous devrez aussi traiter l'extension suivante :

Les exceptions. L'extension avec les exceptions est donnée dans la page décrivant la spécification de fouine. Référez-vous également aux *"notes sur le traitement des exceptions"* disponibles depuis la page du portail des études, rubrique *"Fouine - ressources"*.

NB : il n'est pas demandé de traiter des cas de filtrage comme `with E 2 -> ..`, on se contentera de `E x` (quitte à mettre après des `if x=2 then..  else..`).

On n'a pas le droit d'implémenter les exceptions en utilisant les exceptions de Caml ; il faut donc réfléchir à comment faire tourner les exceptions dans l'interprète. Pour ce faire, soyez conscients du fait que les exceptions permettent de "désactiver un calcul" : lorsque l'on exécute `2+(raise (E 12))`, la partie "`2+·`" doit être jetée à la poubelle (et même phénomène dans le cas où l'on est au sein d'un appel à une fonction).

Référez-vous aux notes de cours sur les exceptions et les continuations, disponibles sur la page du portail des études, pour des conseils. Vous pouvez bien sûr interagir avec les enseignant.e.s pour comprendre comment implémenter les exceptions.

Organisation. Du point de vue de la planification de votre travail, il est indispensable d'avancer pas à pas, ce qui signifie ici extension par extension. On vous encourage dans cette optique à affecter une extension à une personne du binôme, plutot que d'avoir quelqu'un qui ne voit que le parser pendant tout le projet, tandis que l'autre ne voit jamais le parser (et symétriquement pour l'évaluateur).

3 Avancés

Commencez par traiter la partie **"Exceptions"** du rendu des Intermédiaires. Traitez les exceptions *directement*, au sens où vous n'avez pas le droit de vous appuyer sur la réponse à ce qui suit (transformation de programmes) pour exécuter des programmes avec exceptions. Vous pouvez toutefois décider d'utiliser des continuations pour exécuter des exceptions (est-ce clair ?... la fonction `eval` peut utiliser des exceptions, sans que le programme en entrée soit traduit avec une CPS).

Transformations de programmes : passage de continuations.

Le but est d'implémenter la transformation de programmes vue en cours. L'approche est la suivante :

(*i*) On récupère un programme fouine.

Ce programme pourra comporter toutes les constructions vues jusqu'ici, exceptions incluses.

(*ii*) On applique la transformation, de manière à obtenir un programme fouine sans extension.

Référez-vous à ce qui a été décrit en cours, pour implémenter la transformation qui élimine les constructions propres aux exceptions afin d'obtenir un programme fouine[1].

Les fonctions qui font cette transformation utilisent l'extension avec les couples/tuples que vous avez programmée.

Il est fortement conseillé d'écrire sur papier la transformation *avant de la programmer*. Tâchez aussi de programmer de manière claire (le code implémentant la transformation est volontiers touffu si on ne réfléchit pas assez).

Le résultat de la transformation est une fonction qui s'appellera obligatoirement, par convention, `main_transform`. Cette fonction attend en entrée *un couple de continuations*, constitué de la continuation "normale" et de la continuation "exceptionnelle". On pourra lancer l'exécution en évaluant `main_transform (fun z -> z, fun t -> t)` (par exemple).

(*iii*) On peut faire ce qu'on veut du programme résultant : l'exécuter grâce à l'interprète fouine, ou l'afficher : (cf. ci-dessous les options de l'exécutable). Veillez à *bien respecter les consignes données*, afin que l'on puisse tester les programmes que vous rendez.

Transformation de programmes : cps *(continuation passing style)*

`-cps` Applique la transformation pour éliminer les exceptions.

`-outcode` Cette option, combinée avec l'option `-cps`, aura pour effet d'afficher à l'écran le programme résultant de la transformation, sans l'exécuter (par exemple : `./fouine -cps -outcode pgm.ml`).

On rappelle que la fonction ainsi définie doit s'appeler `main_transform`.

`-run` Cette option, combinée avec l'option `-cps`, aura pour effet d'exécuter avec `fouine` le programme résultant de la traduction. Pour ce faire, il faut appliquer la fonction `main_transform` aux arguments initiaux par défaut (un couple avec l'identité deux fois), et puis évaluer.

`-autotest` Cette option, combinée avec l'option `-cps`, aura pour effet de comparer différentes exécutions du programme fourni en entrée. Dans l'ordre :

1. Si l'exécution du programme en entrée par `fouine` fournit la même sortie que l'exécution par Caml du programme en entrée, on affiche `OK`, sinon on affiche `NO`.

   Après quoi on va à la ligne.

[1] 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)) , kE)
⟦let x = e1 in e2⟧ = fun (k,kE) -> ⟦e1⟧(fun x -> ⟦e2⟧(k,kE) )
```

2. Si l'exécution par `fouine` du programme en entrée fournit la même sortie que l'exécution par Caml du programme transformé, on affiche `OK`, sinon on affiche `NO`.

   Après quoi on va à la ligne.

3. Si l'exécution par `fouine` du programme en entrée fournit la même sortie que l'exécution par `fouine` du programme transformé, on affiche `OK`, sinon on affiche `NO`.

   Après quoi on va à la ligne.

Une exécution où tout se passe bien engendrera donc la sortie

```
OK
OK
OK
```

Vous pouvez enrichir l'option `-debug` de manière à ce que des messages informatifs soient affichés lorsque l'une des étapes ci-dessus affiche `NO`.

Ci-dessus, exécuter le programme traduit signifie évaluer l'application de `main_transform` aux arguments par défaut. Pour exécuter les programmes en Caml, il faut fournir un fichier "prélude" pour que Caml connaisse `prInt` et l'exception `E`.

`-optim` Cette option améliore la traduction CPS en appliquant les simplifications décrites dans les notes de cours.

On veillera à programmer un comportement spécifique de l'option `-autotest` si on utilise les deux options à la fois (`autotest` et `optim`, en plus bien sûr de `cps`).

Il est demandé que votre exécutable accepte un appel avec l'option `-cps`, éventuellement accompagnée de `-optim`, combiné avec une option parmi (`-outcode, -run, -autotest`). Dans les autres situations, faites comme bon vous semble (protestez, ne faites rien, évaluez la fonction d'Ackermann...).

Comme toujours, avancez progressivement, avec des étapes intermédiaires : ne traitez que le langage qui est au menu débutants du rendu 2, et mettez en place toute la machinerie des options. Quand tout est propre, ajoutez progressivement tous les éléments de votre interprète.

4 Spécification : exécutable, options, etc.

Options. Référez-vous à ce qui précède (et, le cas échéant, au sujet du rendu 2).

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 aux rendus 1 et 2 : l'esprit est le même, en ce qui concerne la propreté du rendu, les explications, etc. N'oubliez pas d'expliquer dans le README comment vous avez procédé, notamment si vous traitez la partie sur les transformations de programmes.