Projet 2 L3IF : interpréter, compiler

Rendu 2 : pour le 8 mars 2021 à 23h59

(trois 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 que le code lui demande d'afficher.

Pour le moment, fouine n'est pas typé : on peut exécuter des programmes qui seraient refusés par Caml pour des raisons de typage, ce qui peut conduire à des catastrophes...

Utilisation de lex et yacc. Pour la majorité d'entre vous, la programmation des analyses lexicale et syntaxique (outils ocamllex et ocamlyacc) est une nouveauté. C'est un aspect pratique, qui s'apprend en programmant et en faisant des erreurs, mais avant de vous lancer, consultez les ressources dont vous disposez, et notamment les deux archives disponibles depuis la page du cours sur le portail des études : lisez les fichiers README, faites de petites manipulations avec les programmes pour voir comment ça marche.

Il faut que le code que vous rendez se compile *sans conflit :* pas de shift/reduce conflict, pas de reduce/reduce conflict (ces conflits apparaissent lorsque la compilation traite de l'analyse syntaxique).

2 Débutants

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

1. les expressions arithmétiques ;

   Il vous est recommandé d'utiliser l'un des deux exemples pour lex/yacc disponibles sur la page www du cours comme point de départ.

2. les `let...in`, dans la version de base, et avec les deux extensions "au-delà du cœur"

   - on vous conseille de commencer par avoir une version qui marche sans accepter les écritures alternatives pour les `let.. in` ;

     à noter qu'il serait maladroit que le type pour les expressions fouine comporte plusieurs constructeurs correspondant aux différentes écritures d'un `let.. in` ;
   - cf. les transparents pour l'exécution du `let.. in`, avec les environnements (il est *obligatoire* d'utiliser des environnements).

3. le `if then else` ;

4. la "fonction" `prInt`.

   À noter que vous n'avez pas encore les fonctions dans le langage, donc la seule manière de faire un appel de fonction est pour le moment en faisant `prInt e` : vous pouvez avoir une règle *ad hoc* pour cela dans votre grammaire.

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 ;
   - plusieurs points en lien avec l'analyse lexicale et syntaxique pour le traitement des fonctions et de l'application :
     - Comme pour les `let .. in` ci-dessus, vous pouvez commencer par avoir une version qui marche et n'accepte qu'une écriture pour les fonctions, pour ensuite traiter les extensions attendues.
       Rappelons qu'il ne fait pas sens d'avoir un traitement spécifique pour les "fonctions à plusieurs arguments". En fouine, tout se ramène à des fonctions à un seul argument.
     - Comprenez comment écrire la grammaire qui permet d'analyser les applications, qui ne sont notées qu'avec des espaces (comme par exemple dans `(fun x y z -> BLABLA) t (56+u) v`).
       Pour ce faire, il faut dire qu'on peut écrire des choses comme `e1 e2 e3 e4`, où `e1` est la fonction, et `e2, e3, e4` sont "ses arguments". En écrivant cela, on a l'idée que `e1` est une "expression simple" : ça ne peut pas être une application. `e2` (et pareil pour `e3, e4`) est une expression quelconque, à ceci près que si c'est une application, il faut nécessairement parenthéser, sinon la signification change.
       Ces considérations devraient vous suggérer comment structurer votre grammaire de manière à traiter l'application comme en Caml.
     - Traitez la difficulté en lien 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. Avec l'ajout des références, la notion de valeur est modifiée. Dans un premier temps, on pourra se contenter d'utilisations élémentaires de `ref`, qui ne sera pas traitée comme une "vraie fonction"[1]. À terme, il serait bon que `prInt` et `ref` soient vues comme des fonctions et puissent être passées en argument à des fonctions.

   À noter qu'en présence d'aspects impératifs, *l'ordre d'évaluation compte* (que ce soit dans le traitement des opérateurs binaires ou dans le traitement de l'application) : à vous de concevoir des tests permettant de vérifier que votre interprète se comporte comme Caml.

   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 (sans doute un gros tableau).

[1] Alors qu'en Caml :
```
# ref;;
- :  'a -> 'a ref = <fun>
```

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`.

Toujours à propos de la séquence, vous pouvez réfléchir *tôt* aux difficultés que peut poser la présence de deux usages pour `;` : opérateur de séquence, et séparateur pour les listes.

2. **Les couples.** On ajoute la virgule, pour faire des couples.

   (a) **Version "bon marché".** On impose d'utiliser des parenthèses autour des couples, tant au moment de leur définition `(3,5)` qu'au moment de leur "déconstruction" `let (x,y) = c in ...`.

   (b) **Version propre.** Introduisez la notion de *motif*, et traitez les couples par ce biais. Autorisez des écritures comme `let t = 3, f y in..` ou `let x,y = c in..`, sans que les parenthèses soient obligatoires.

   Extensions optionnelles : les tuples (p.ex. `(3,4,2)`), l'écriture `match c with | (x,y) -> ...`.

3. **Les listes.** Ajoutez `[], ::`, et le filtrage (avec `match...with` et avec `function`). L'interprète pourra engendrer un `Warning` si un filtrage ne comporte qu'un cas sur deux.

NB : si vous avez une dextérité raisonnable pour la manipulation de lex et yacc, vous pouvez, au lieu d'ocamllex/ocamlyacc, vous intéresser à menhir, un outil spécifique à Caml qui améliore le confort vis-à-vis de lex et yacc.

5 Spécification : exécutable, options, tests, README

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 que le code lui dit d'afficher (à l'aide de `prInt`), *et rien de plus* (pas de "messages conviviaux", pas d'affichage pour le debug).

Les débutants pourront se contenter de cela pour le rendu 2, mais regardez les autres options si vous avez le temps, car il faudra avoir les options pour le rendu 3.

`-showsrc` Cette option aura pour effet d'afficher le programme écrit en entrée (cf. paragraphe "Affichage" dans la description du langage fouine) *uniquement :* on désactive l'affichage via `prInt`.

`-debug` Cette option aura pour effet d'afficher à la fois ce qui s'affiche avec l'option `-showsrc` et ce qui s'affiche via le programme.

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

Un exemple de petit programme Caml est disponible sur la page www du cours pour vous aider à gérer l'exécutable et les options (pas besoin de tout comprendre dans l'exemple, il suffit de vous en inspirer pour programmer l'option `-debug`).

Tests.

- Un certain nombre de fichiers de tests vous seront proposés à partir de la page www du cours. Vos programmes doivent tourner correctement sur ces tests (c'est un problème si vous rendez un programme qui échoue sur ces tests).

- Fournissez un répertoire avec des programmes de test supplémentaires que vous avez soumis à `fouine` (inutile d'y inclure ceux que nous vous fournissons). Veillez à ce que les tests que vous concevez couvrent l'ensemble des aspects que vous traitez (lex-yacc, exécution des divers composants du langage fouine).

  Il est recommandé de concevoir les tests au fur et à mesure que vous étendez votre interprète, et pas à la fin.

- 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".

Avancez avec des échéances intermédiaires (cf. ci-dessous) : il vaut mieux traiter proprement "tout sauf XXX" que d'avoir tout qui marche mal.

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 moment de la date limite.

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

- pour les binômes Débutants

  1. mise en place du code, et traitement des expressions arithmétiques sans `let.. in` (tout traiter : parser, interprète, tests, README) ;
  2. ajout de `let.. in` (et donc des environnements) ;
  3. le reste, et peaufinages finaux

  Une remarque de bon sens, qui vaut pour tout le monde, pas que les Débutants : si la moitié du binôme travaille sur l'analyse syntaxique, l'autre moitié ne doit pas dépendre de la finalisation de ce travail. On se met d'accord sur le type des valeurs que renvoie l'analyse syntaxique, et puis on se fabrique des exemples "à la main" pour pouvoir tester l'interprète en attendant de pouvoir brancher l'analyse syntaxique.

- pour les Intermédiaires

  1. le menu des débutants ;
  2. les fonctions qui marchent, sans les fioritures ;
     (sans les fioritures = une fonction s'écrit uniquement `fun x -> e`, pas de `let f x = ..` ni de `fun x y z -> ..`)
  3. les fonctions récursives et les fioritures.

- pour les Avancés

  1. Tout jusqu'aux fonctions ;
  2. les aspects impératifs ;
  3. les couples et les listes (ou juste les couples si vous manquez de temps).

À chaque étape intermédiaire, veillez à aboutir 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).