Travail pratique #1

IFT-2035

4 octobre 2022

1 Survol

Ce TP vise à améliorer la compréhension des langages fonctionnels en utilisant un langage de programmation fonctionnel (Haskell) et en écrivant une partie d'un interpréteur d'un langage de programmation fonctionnel (en l'occurence une sorte de Lisp). Les étapes de ce travail sont les suivantes :

1. Parfaire sa connaissance de Haskell.
2. Lire et comprendre cette donnée. Cela prendra probablement une partie importante du temps total.
3. Lire, trouver, et comprendre les parties importantes du code fourni.
4. Compléter le code fourni.
5. Écrire un rapport. Il doit décrire **votre** expérience pendant les points précédents : problèmes rencontrés, surprises, choix que vous avez dû faire, options que vous avez sciemment rejetées, etc... Le rapport ne doit pas excéder 5 pages.

Ce travail est à faire en groupes de deux. Le rapport, au format LaTeX exclusivement (compilable sur `ens.iro`) et le code sont à remettre par remise électronique avant la date indiquée. Aucun retard ne sera accepté. Indiquez clairement votre nom au début de chaque fichier.

Ceux qui veulent faire ce travail seul(e)s doivent d'abord en obtenir l'autorisation, et l'évaluation de leur travail n'en tiendra pas compte. Des groupes de 3 ou plus sont **exclus**.

$e ::= n$	Un entier signé en décimal
$\mid x$	Une variable
$\mid (e_0\ e_1\ ...\ e_n)$	Un appel de fonction (*curried*)
$\mid (\mathsf{fn}\ (x_1\ ...\ x_n)\ e)$	Une fonction ; les arguments sont *curried*
$\mid (\mathsf{let}\ (d_1\ ...\ d_n)\ e)$	Ajout de déclarations locales
$\mid + \mid - \mid * \mid /$	Opérations arithmétiques prédéfinies
$\mid (\mathsf{add}\ e_1\ e_2) \mid \mathsf{nil} \mid (\mathsf{list}\ e_1\ ...\ e_n)$	Construction de listes
$\mid (\mathsf{match}\ e\ b_1\ ...\ b_n)$	Filtrage sur les listes
$b ::= (\mathsf{nil}\ e) \mid ((\mathsf{add}\ x_1\ x_2)\ e)$	Branche de filtrage
$d ::= (x\ e)$	Déclaration de variable
$\mid ((x\ x_1\ ...\ x_n)\ e)$	Déclaration de fonction

FIGURE 1 – Syntaxe de Slip

2 Slip : Une sorte de Lisp

Vous allez travailler sur l'implantation d'un langage fonctionnel dont la syntaxe est inspirée du langage Lisp. La syntaxe de ce langage est décrite à la Figure 1. À remarquer que comme toujours avec la syntaxe de style Lisp, les parenthèses sont *significatives*.

La fonction prédéfinie add construit un élément de liste (comme le (:) de Haskell) ; nil est une constante prédéfinie utilisée pour représenter la liste vide (comme [] en Haskell).

La forme let est utilisée pour donner des noms à des définitions locales. Exemple :

$$(\mathsf{let}\ ((x\ 2)\ (y\ 3))\ (+\ x\ y)) \quad \rightsquigarrow^* \quad 5$$

Vu que beaucoup de définitions locales sont des fonctions, la forme let accepte une syntaxe particulière pour définir des fonctions :

$$(\mathsf{let}\ ((y\ 10)\ ((\mathit{div2}\ x)\ (/\ x\ 2)))\ (\mathit{div2}\ y)) \quad \rightsquigarrow^* \quad 5$$

Les définitions d'un let peuvent être récursives et même mutuellement récursives. Exemple :

$$(\mathsf{let}\ (((\mathit{even}\ \mathit{xs})\ (\mathsf{match}\ \mathit{xs}\ (\mathsf{nil}\ 0)\ ((\mathsf{add}\ x\ \mathit{xs})\ (\mathit{odd}\ \mathit{xs}))))\ ((\mathit{odd}\ \mathit{xs})\ (\mathsf{match}\ \mathit{xs}\ (\mathsf{nil}\ 1)\ ((\mathsf{add}\ x\ \mathit{xs})\ (\mathit{even}\ \mathit{xs})))))\ (\mathit{odd}\ (\mathsf{list}\ 2\ 3\ 4))) \quad \rightsquigarrow^* \quad 0$$

2.1 Sucre syntaxique

Les fonctions n'ont en réalité qu'un seul argument : la syntaxe offre la possibilité de déclarer et de passer plusieurs arguments, mais ce n'est que du sucre syntaxique pour des définitions et des appels en forme *curried*. Dans le même ordre d'idée, le constructeur list est une forme simplifiée équivalente à une combinaison de add et de nil. Plus précisément, les équivalences suivantes sont vraies pour les expressions :

$$\begin{array}{rcl} (e_0\ e_1\ e_2\ ...\ e_n) & \iff & (..((e_0\ e_1)\ e_2)\ ...\ e_n) \\ (\mathsf{fn}\ (x_1\ ...\ x_n)\ e) & \iff & (\mathsf{fn}\ (x_1)\ ...\ (\mathsf{fn}\ (x_n)\ e)..) \\ (\mathsf{list}\ e_1\ ...\ e_n) & \iff & (\mathsf{add}\ e_1\ ...\ (\mathsf{add}\ e_n\ \mathsf{nil})..) \end{array}$$

De plus, la syntaxe d'une déclaration de fonction est elle aussi du sucre syntaxique, et elle est régie par l'équivalence suivante pour les *déclarations* :

$$((x\ x_1\ ...\ x_n)\ e) \iff (x\ (\mathsf{fn}\ (x_1\ ...\ x_n)\ e))$$

Votre première tâche sera d'écrire une fonction *s2l* qui va "éliminer" le sucre syntaxique, c'est à dire faire l'expansion des formes de gauche (présumément plus pratiques) dans leur équivalent de droite, de manière à réduire le nombre de cas différents à gérer dans le reste de l'implantation du langage. Cette fonction va aussi transformer le code dans un format plus facile à manipuler par la suite.

2.2 Sémantique dynamique

Slip, comme Lisp, est un langage typé dynamiquement, c'est à dire que ses variables peuvent contenir des valeurs de n'importe quel type. Il n'y a donc pas de sémantique statique (règles de typage).

Les valeurs manipulées à l'exécution par notre langage sont les entiers, les fonctions, et les listes (dénotées [], et $[v_1\ .\ v_2]$). De plus la notation de listes est étendue de sorte que $[v_1\ .\ [v_2\ .\ [v_3\ .\ v_4]]]$ se note $[v_1\ v_2\ v_3\ .\ v_4]$ et qu'un ".[]" final peut s'éliminer. Donc $[v_1\ v_2]$ est équivalent à $[v_1\ .\ [v_2\ .\ []]]$.

Les règles d'évaluation fondamentales sont les suivantes :

$$\begin{array}{rcl} ((\mathsf{fn}\ (x)\ e)\ v) & \rightsquigarrow & e[v/x] \\ (\mathsf{let}\ ((x_1\ v_1)\ ...\ (x_n\ v_n))\ e) & \rightsquigarrow & e[v_1, ..., v_n/x_1, ..., x_n] \end{array}$$

où la notation $e[v/x]$ représente l'expression e dans un environnement où la variable x prend la valeur v. L'usage de v dans les règles ci-dessus indique qu'il s'agit bien d'une valeur plutôt que d'une expression non encore évaluée. Par exemple le v dans la première règle indique que lors d'un appel de fonction, l'argument doit être évalué avant d'entrer dans le corps de la fonction, i.e. on utilise l'appel par valeur.

En plus des deux règles β ci-dessus, les différentes primitives se comportent

comme suit :

$$\begin{array}{rcl}
(+\ n_1\ n_2) & \rightsquigarrow & n_1 + n_2 \\
(-\ n_1\ n_2) & \rightsquigarrow & n_1 - n_2 \\
(*\ n_1\ n_2) & \rightsquigarrow & n_1 \times n_2 \\
(/\ n_1\ n_2) & \rightsquigarrow & n_1 \div n_2 \\
\mathsf{nil} & \rightsquigarrow & [] \\
(\mathsf{add}\ v_1\ v_2) & \rightsquigarrow & [v_1\ .\ v_2] \\
(\mathsf{match}\ []\ ...\ (\mathsf{nil}\ e_n)\ ...) & \rightsquigarrow & e_n \\
(\mathsf{match}\ [v_1\ .\ v_2]\ ...\ ((\mathsf{add}\ x\ xs)\ e_c)\ ...) & \rightsquigarrow & e_c[v_1, v_2/x, xs]
\end{array}$$

Donc il s'agit d'une variante du λ-calcul, sans grande surprise. La portée est lexicale et l'ordre d'évaluation est présumé être "par valeur", mais vu que le langage est pur, la différence n'est pas très importante pour ce travail.

3 Implantation

L'implantation du langage fonctionne en plusieurs phases :

1. Une première phase d'analyse lexicale et syntaxique transforme le code source en une représentation décrite ci-dessous, appelée *Sexp* dans le code. Ce n'est pas encore tout à fait un arbre de syntaxe abstraite (cela s'apparente en fait à XML).
2. Une deuxième phase, appelée *s2l*, termine l'analyse syntaxique et commence la compilation, en transformant cet arbre en un vrai arbre de syntaxe abstraite dans la représentation appelée *Lexp* dans le code. Comme mentionné, cette phase commence déjà la compilation vu que le langage *Lexp* n'est pas identique à notre langage source. En plus de terminer l'analyse syntaxique, cette phase élimine le sucre syntaxique (i.e. applique les règles de la forme ... $\Longleftrightarrow$ ...), et doit faire quelques ajustements supplémentaire.
3. Une troisième phase, appelée *l2d* élimine les noms de variables, et les remplace par des indexes dans l'environnement, pour une évaluation plus efficace vu qu'il n'y a plus besoin de chercher les variables avec des comparaisons de chaînes de caractères.
4. Finalement, une fonction *eval* procède à l'évaluation de l'expression par interprétation.

Une partie de l'implantation est déjà fournie : la première ainsi que divers morceaux des autres. Votre travail consistera à compléter les trous.

3.1 Analyse lexicale et syntaxique : *Sexp*

L'analyse lexicale et syntaxique est déjà implantée pour vous. Elle est plus permissive que nécessaire et accepte n'importe quelle expression de la forme suivante :

$$e ::= n \mid x \mid \text{`(' } \{\, e \,\} \text{ `)'}$$

n est un entier signé en décimal.
Il est représenté dans l'arbre en Haskell par : Snum n.

x est un symbole qui peut être composé d'un nombre quelconque de caractères alphanumériques et/ou de ponctuation. Par exemple '+' est un symbole, '<=' est un symbole, 'voiture' est un symbole, et 'a+b' est aussi un symbole. Dans l'arbre en Haskell, un symbole est représenté par : Ssym x.

'(' { e } ')' est une liste d'expressions. Dans l'arbre en Haskell, les listes d'expressions sont représentées par des listes simplement chaînées constituées de paires Scons *left right* et du marqueur de fin Snil. *left* est le premier élément de la liste et *right* est le reste de la liste.

Par exemple l'analyseur syntaxique transforme l'expression `(+ 2 3)` dans l'arbre suivant en Haskell :

```
Scons (Ssym "+")
      (Scons (Snum 2)
             (Scons (Snum 3)
                    Snil))
```

L'analyseur lexical considère qu'un caractère ';' commence un commentaire, qui se termine à la fin de la ligne.

3.2 La représentation intermédiaire *Lexp*

Cette représentation intermédiaire est une sorte d'arbre de syntaxe abstraite. Dans cette représentation, +, -, ... sont simplement des variables prédéfinies, et le sucre syntaxique n'est plus disponible, donc les fonctions ne prennent plus qu'un seul argument et la forme let ne peut définir que des variables.

Elle est définie par le type :

```
data Lexp = Lnum Int
          | Lref Var
          | Llambda Var Lexp
          | Lcall Lexp Lexp
          | Lnil
          | Ladd Lexp Lexp
          | Lmatch Lexp Var Var Lexp Lexp
          | Lfix [(Var, Lexp)] Lexp
          deriving (Show, Eq)
```

L'expression conditionnelle (match e (nil e_n) ((add x xs) e_c)) se traduit par Lmatch e x xs e_c e_n. Le Lfix correspond aux expressions let de Slip, mais on l'appelle ici "fix" pour rappeler que cela permet les définitions récursives (par "point fixe").

3.3 L'environnement d'exécution

Le code fourni défini aussi l'environnement initial d'exécution, qui contient les fonctions prédéfinies du langage telles que l'addition, la soustraction, etc. Il est défini comme une table qui associe à chaque identificateur prédéfini la valeur (de type *Value*) associée. La valeur ne sera utilisée que lors de l'évaluation, mais la liste est aussi utile avant, pour détecter l'usage d'une variable non-définie.

3.4 La représentation intermédiaire *Dexp*

Dexp est identique à *Lexp* sauf que les variables sont représentées non pas par des chaînes de caractères mais par des "indexes de *de Bruijn*", du nom du mathématicien Nicolaas Govert de Bruijn qui fut le premier à les utiliser (dans son système appelé Automath). Concrètement cela signifie que les variables sont représentées par un entier qui indique leur distance dans l'environnement : la variable la plus récemment déclarée a index 0, l'antérieure a index 1, etc... Par exemple, l'expression *Lexp* suivante :

```
Llambda "x" (Llambda "y" (Ladd (Lref "x") (Lref "y")))
```

se traduit par l'expression *Dexp* suivante :

```
Dlambda (Dlambda (Dadd (Dref 1) (Dref 0)))
```

Comme vous pouvez le voir, les occurrences "x" et "y" qui correspondent à des définitions/déclarations ont complètement disparu (vu qu'on sait que `Dlambda` introduit exactement une variable), alors que celles qui correspondent à des références sont remplacées par des indexes.

3.5 Évaluation

L'évaluateur utilise l'environnement initial pour réduire une expression (de type *Dexp*) à une valeur (de type *Value*). Grâce à l'usage de *Dexp*, l'évaluateur n'a plus à se préoccuper de noms de variables.

4 Cadeaux

Comme mentionné, l'analyseur lexical et l'analyseur syntaxique sont déjà fournis. Dans le fichier `slip.hs`, vous trouverez les déclarations suivantes :

Sexp est le type des arbres, il défini les différents noeuds qui peuvent y apparaître.

readSexp est la fonction d'analyse syntaxique.

showSexp est un pretty-printer qui imprime une expression sous sa forme "originale".

Lexp est le type de la représentation intermédiaire du même nom.

s2l est la fonction qui transforme une expression de type *Sexp* en *Lexp*.

Value est le type du résultat de l'évaluation d'une expression.

env0 est l'environnement initial.
Dexp est le type de la représentation intermédiaire du même nom.
l2d est la fonction qui transforme une expression de type *Lexp* en *Dexp*.
eval est la fonction d'évaluation qui transforme une expression de type *Dexp* en une valeur de type *Value*.
evalSexp est une fonction qui combine les phases ci-dessus pour évaluer une *Sexp*.
run est la fonction principale qui lie le tout ; elle prend un nom de fichier et applique *evalSexp* sur toutes les expressions trouvées dans ce fichier.

Voilà ci-dessous un exemple de session interactive sur une machine GNU/Linux, avec le code fourni :

```
% ghci
GHCi, version 9.0.2: https://www.haskell.org/ghc/  :? for help
ghci> :load "slip.hs"
[1 of 1] Compiling Main             ( slip.hs, interpreted )

slip.hs:258:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘l2d’:
        Patterns not matched:
            [] (Lref _)
            [] (Llambda _ _)
            [] (Lcall _ _)
            [] Lnil
            ...
    |
258 | l2d _ (Lnum n) = Dnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^

slip.hs:268:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns not matched:
            [] (Dref _)
            [] (Dlambda _)
            [] (Dcall _ _)
            [] Dnil
            ...
    |
268 | eval _ (Dnum n) = Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^
Ok, one module loaded.
ghci> run "exemples.slip"
[2,*** Exception: slip.hs:258:1-23: Non-exhaustive patterns in function l2d
```

```
ghci>
Leaving GHCi.
```

Les avertissements et l'exception levée sont dus au fait que le code a besoin de vos soins. Le code que vous soumettez ne devrait pas souffrir de tels avertissements, et l'appel à `run` devrait renvoyer la liste des valeurs décrites en commentaires dans le fichier.

5 À faire

Vous allez devoir compléter l'implantation de ce langage, c'est à dire compléter *s2l*, *l2d*, et *eval*. Je recommande de le faire "en largeur" plutôt qu'en profondeur : compléter les fonctions peu à peu, pendant que vous avancez dans `exemples.slip` plutôt que d'essayer de compléter tout *s2l* avant de commencer à attaquer la suite. Ceci dit, libre à vous de choisir l'ordre qui vous plaît.

De même je vous recommande fortement de travailler en binôme (*pair programming*) plutôt que de vous diviser le travail, vu que la difficulté est plus dans la compréhension que dans la quantité de travail.

Le code contient des indications des endroits que vous devez modifiez. Généralement cela signifie qu'il ne devrait pas être nécessaire de faire d'autres modifications, sauf ajouter des fonctions auxiliaires. Vous pouvez aussi modifier le reste du code, si vous le voulez, mais il faudra alors justifier ces modifications dans votre rapport en expliquant pourquoi cela vous a semblé nécessaire.

Vous devez aussi fournir un fichier de tests `tests.slip`, similaire à `exemples.slip`, mais qui contient au moins 5 tests que *vous* avez écrits.

5.1 Remise

Pour la remise, vous devez remettre trois fichiers (`slip.hs`, `tests.slip`, et `rapport.tex`) par la page Moodle (aussi nommé StudiUM) du cours. Assurez-vous que le rapport compile correctement sur `ens.iro` (auquel vous pouvez vous connecter par SSH).

6 Détails

- La note sera divisée comme suit : 25% pour le rapport, 60% pour le code (réparti entre *s2l*, `l2d`, et *eval*), et 15% pour les tests.
- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, ...) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat.
- Le code ne doit en aucun cas dépasser 80 colonnes.
- Vérifiez la page web du cours, pour d'éventuels errata, et d'autres indications supplémentaires.
- La note est basée d'une part sur des tests automatiques, d'autre part sur la lecture du code, ainsi que sur le rapport. Le critère le plus important,

et que votre code doit se comporter de manière correcte. Ensuite, vient la qualité du code : plus c'est simple, mieux c'est. S'il y a beaucoup de commentaires, c'est généralement un symptôme que le code n'est pas clair ; mais bien sûr, sans commentaires le code (même simple) est souvent incompréhensible. L'efficacité de votre code est sans importance, sauf si votre code utilise un algorithme vraiment particulièrement ridiculement inefficace.