Travail pratique #1

IFT-2035

2 octobre 2024

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 2 étudiants. 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
	$\mid$ (if e e_{then} e_{else})	Expression conditionelle
	$\mid (e_0\ e_1\ ...\ e_n)$	Un appel de fonction
	$\mid$ (fob $(x_1\ ...\ x_n)$ e)	Une fonction
	$\mid$ (let x e_1 e_2)	Déclaration locale simple
	$\mid$ (fix $(d_1\ ...\ d_n)$ e)	Déclarations locales récursives
	$\mid + \mid - \mid * \mid / \mid ...$	Opérations binaires prédéfinies
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(!).

Les formes let et fix (nommée ainsi parce qu'elle construit ce qu'on appelle parfois un "point fixe") sont utilisées pour donner des noms à des définitions locales. La différence est que let est une forme plus simple qui n'autorise pas les définitions récursives. Exemple :

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

$$\begin{array}{l}(\mathsf{fix}\ ((x\ 2)\\ \qquad (y\ 3))\\ \quad (+\ x\ y))\end{array} \quad \rightsquigarrow^* \quad 5$$

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

$$\begin{array}{l}(\mathsf{fix}\ ((y\ 10)\\ \qquad ((\mathit{div2}\ x)\\ \qquad\ (/\ x\ 2)))\\ \quad (\mathit{div2}\ y))\end{array} \quad \rightsquigarrow^* \quad 5$$

Les définitions d'un fix peuvent être mutuellement récursives. Exemple :

$$\begin{array}{l}(\mathsf{fix}\ (((\mathit{even}\ x)\ (\mathsf{if}\ (=\ x\ 0)\ \mathit{true}\ (\mathit{odd}\ (-\ x\ 1))))\\ \qquad ((\mathit{odd}\ x)\ (\mathsf{if}\ (=\ x\ 0)\ \mathit{false}\ (\mathit{even}\ (-\ x\ 1)))))\\ \quad (\mathit{odd}\ 42))\end{array} \quad \rightsquigarrow^* \quad \mathsf{False}$$

2.1 Sucre syntaxique

La syntaxe d'une déclaration de fonction est du sucre syntaxique, et elle est régie par l'équivalence suivante pour les déclarations :

$$((x\ x_1\ ...\ x_n)\ e) \quad \Longleftrightarrow \quad (x\ (\mathsf{fob}\ (x_1\ ...\ x_n)\ e))$$

où fob s'appelle ainsi parce qu'il construit un objet fonction (*function object* en anglais). 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 pour le programmeur) 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 :

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

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 les v dans la première règle indiquent que lors d'un appel de fonction, les arguments doivent être évalués avant d'entrer dans le corps de la fonction, i.e. on veut l'appel par valeur.

En plus des règles β ci-dessus, les différentes primitives se comportent comme suit :

$$\begin{array}{rcl}
(+\ n_1\ n_2) & \rightsquigarrow & n_1 + n_2 \\
(-\ n_1\ n_2) & \rightsquigarrow & n_1 - n_2 \\
(*\ n_1\ n_2) & \rightsquigarrow & n_1 \times n_2 \\
(/\ n_1\ n_2) & \rightsquigarrow & n_1 \div n_2 \\
(\mathsf{if}\ \mathit{true}\ e_1\ e_2) & \rightsquigarrow & e_1 \\
(\mathsf{if}\ \mathit{false}\ e_1\ e_2) & \rightsquigarrow & e_2
\end{array}$$

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 un arbre de syntaxe abstraite (cela s'apparente en fait à XML ou JSON).
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. les règles de la forme ... $\Longleftrightarrow$ ...), et doit faire quelques ajustements supplémentaire.
3. 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

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 `Snode` *left right* et du marqueur de fin `Snil`. *left* est le premier élément de la liste et *right* est le reste de la liste (i.e. ce qui le suit).

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

```
Snode (Ssym "+") [Snode (Snum 2), Snode (Snum 3)]
```

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 fix ne peut définir que des variables.

Elle est définie par le type :

```
data Lexp = Lnum Int             -- Constante entière.
          | Lbool Bool           -- Constante Booléenne.
          | Lvar Var             -- Référence à une variable.
          | Ltest Lexp Lexp Lexp -- Expression conditionelle.
          | Lfob [Var] Lexp      -- Construction de fobjet.
          | Lsend Lexp [Lexp]    -- Appel de fobjet.
          | Llet Var Lexp Lexp   -- Déclaration non-récursive.
          -- Déclaration d'une liste de variables qui peuvent être
          -- mutuellement récursives.
          | Lfix [(Var, Lexp)] Lexp
          deriving (Show, Eq)
```

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 peut aussi être utile avant, pour détecter l'usage d'une variable non-définie.

3.4 Évaluation

L'évaluateur utilise l'environnement initial pour réduire une expression (de type *Lexp*) à une valeur (de type *Value*). Bien sûr, l'environnement initial s'appelle ainsi parce que c'est la valeur initiale de l'environnement, mais au cours de l'évaluation, cet environnement sera ajusté en y ajoutant les variables locales et autres arguments de fonction.

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.
eval est la fonction d'évaluation qui transforme une expression de type *Lexp* 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 slip.hs
GHCi, version 9.0.2: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( slip.hs, interpreted )

slip.hs:241:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns not matched:
            [] (Lbool _)
            [] (Lvar _)
            [] (Ltest _ _ _)
            [] (Lfob _ _)
            ...
    |
241 | eval _ (Lnum n) = Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^
Ok, one module loaded.
ghci> run "exemples.slip"
[2,*** Exception: slip.hs:241:1-24: Non-exhaustive patterns in function eval

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 d'exemples.

5 À faire

Vous allez devoir compléter l'implantation de ce langage, c'est à dire compléter *s2l* et *eval*.

Vous devez aussi fournir un fichier `tests.slip`, similaire à `exemples.slip`, mais qui contient au moins 5 tests que *vous* avez écrits (avec en commentaire la valeur de retour que vous pensez devrait être renvoyée). Les tests sont évalués sur les critères suivants :

— Ils doivent bien sûr être corrects.

— Ils doivent être suffisamment différents les uns des autres (et des exemples fournis) pour détecter des erreurs différentes.
— Votre implantation de Slip doit les exécuter correctement.

5.1 Recommendations

Je recommande de faire ce travail "en largeur" plutôt qu'en profondeur : compléter les fonctions peu à peu, pendant que vous avancez dans des exemples de code Slip simples 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 (`https://fr.wikipedia.org/wiki/Programmation_en_bin%C3%B4me`) plutôt que de vous diviser le travail, vu que la difficulté est plus dans la compréhension que dans la quantité de code.

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.

5.2 Remise

Pour la remise, vous devez remettre 3 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 : 30% pour *s2l*, 25% pour le rapport, 15% pour les tests et 30% pour *eval*.
— Tout usage de matériel (code, 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 est 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.