Travail pratique #1

IFT-2035

May 19, 2025

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 ne peuvent pas faire ce travail en groupe 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
	$\mid (e_1\ e_2\ ...\ e_n)$	Un appel de fonction (*currifié*)
	$\mid (\mathsf{abs}\ (x_1\ ...\ x_n)\ e)$	Une abstraction (*currifée*)
	$\mid (\mathsf{def}\ (d_1\ ...\ d_n)\ e)$	Ajout de déclarations locales
	$\mid + \mid - \mid * \mid / \mid ...$	Opérations arithmétiques prédéfinies
	$\mid (\mathsf{if}\ e\ e_t\ e_e)$	Expression conditionelle
	$\mid (\mathsf{new}\ c\ e_1\ ...\ e_n)$	Appel de constructeur
	$\mid (\mathsf{filter}\ e\ b_1\ ...\ b_n)$	Filtrage sur donnée
$b ::= (\_\ e) \mid (c\ e) \mid ((c\ x_1\ ...\ x_e)\ e)$		Branche de filtrage
$d ::= (x\ e)$		Déclaration de variable
	$\mid (x\ (x_1\ ...\ x_n)\ e)$	Déclaration de fonction

Figure 1: Syntaxe de Psil

2 Psil: 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. De même, en accord avec la tradition de Lisp, la notation est toujours de style préfixe.

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

```
(def ((x 2)
      (y 3))      ⇝*  5
 (+ x y))
```

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

```
(def ((y 10)
      (div2 (x)
       (/ x 2)))   ⇝*  5
 (div2 y))
```

Tout comme les définitions de Haskell, les définitions d'un def peuvent être (mutuellement) récursives. Exemple:

```
(def ((even (xs)
       (filter xs (nil: 0) ((cons: x xs) (odd xs))))
      (odd (xs)                                          ⇝*  1
       (filter xs (nil: 1) ((cons: x xs) (even xs)))))
 (odd (new cons: 2 (new cons: 3 (new nil:)))))
```

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*. Plus précisément, les équivalences suivantes sont vraies pour les *expressions*:

$$\begin{array}{rcl} (e_1\ e_2\ e_3\ ...\ e_n) & \iff & (..((e_1\ e_2)\ e_3)\ ...\ e_n) \\ (\mathsf{abs}\ (x_1\ ...\ x_n)\ e) & \iff & (\mathsf{abs}\ (x_1)\ ...\ (\mathsf{abs}\ (x_n)\ e)..) \\ (\mathsf{if}\ e\ e_t\ e_e) & \iff & (\mathsf{filter}\ e\ (\mathit{true}\colon e_t)\ (\mathit{false}\colon e_e)) \end{array}$$

La syntaxe d'une déclaration de fonction est elle aussi du sucre syntaxique, et elle est régie par les équivalences suivantes sur les déclarations:

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

Il y a finalement une règle supplémentaire pour les constructeurs qui n'ont pas d'arguments:

$$(c\ e) \iff ((c)\ e) \qquad \text{Branche de filtrage}$$

Tous ces cas de sucre syntaxique sont éliminés en faisant 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. Cela sera fait dans la fonction *s2l*.

2.2 Sémantique dynamique

Les valeurs manipulées à l'exécution par notre langage sont les entiers, les fonctions, et les valeurs construites avec new (dénotées $[c\ v_1\ ...\ v_n]$).

Les règles d'évaluation fondamentales sont les suivantes:

$$\begin{array}{rcl} ((\mathsf{abs}\ (x)\ e)\ v) & \rightsquigarrow & e[v/x] \\ (\mathsf{def}\ ((x_1\ \tau_1\ v_1)\ ...\ (x_n\ \tau_n\ v_n))\ e) & \rightsquigarrow & e[v_1,..,v_n/x_1,..,x_n] \end{array}$$

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 devrait 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:

$$\begin{array}{rcl} (+\ n_1\ n_2) & \rightsquigarrow & n_1 + n_2 \\ (-\ n_1\ n_2) & \rightsquigarrow & n_1 - n_2 \\ (*\ n_1\ n_2) & \rightsquigarrow & n_1 \times n_2 \\ (/\ n_1\ n_2) & \rightsquigarrow & n_1 \div n_2 \\ (\mathsf{new}\ c\ v_1\ ...\ v_n) & \rightsquigarrow & [c\ v_1\ ...\ v_n] \\ (\mathsf{filter}\ [c\ v_1\ ...\ v_n]\ ...\ ((c\ x_1\ ...\ x_n)\ e_i)...) & \rightsquigarrow & e_i[v_1,..,v_n/x_1,..,x_n] \end{array}$$

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 S (ou *Sexp*) dans le code, qui est une sorte d'arbre de syntaxe abstraite.

2. Une deuxieme phase, appelée *s2l*, termine l'analyse syntaxique et commence la compilation, en transformant cet arbre en un autre arbre de syntaxe abstraite dans la représentation appellée L (ou *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 fait 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: les deux premières ainsi que des morceaux simples des 2 autres. Votre travail consistera à compléter *s2l*, et *eval*.

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:

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

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 début Snil. *right* est le dernier élément de la liste et *left* est le reste de la liste (i.e. ce qui le précède).

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

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

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 def ne peut définir que des variables.

Elle est définie par le type:

```
type Var = String
type Constructor = Var
type Lpat = Maybe (Constructor, [Var])

data Lexp = Lnum Int
          | Lvar Var
          | Labs Var Lexp
          | Lapply Lexp Lexp
          | Lnew Constructor [Lexp]
          | Lfilter Lexp [(Lpat, Lexp)]
          | Ldef [(Var, Lexp)] Lexp
          deriving (Show, Eq)
```

C'est sur cette représentation que votre travail va se concentrer.

4 Cadeaux

Comme mentionné, beaucoup de code est déjà fourni. Dans le fichier `psil.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 L, et *Ltype* le type des types de L.

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 qui donne la valeur des variables prédéfinies.

eval est la fonction d'évaluation qui transforme une expression de type *Lexp* en une valeur de type *Value*.

run est la fonction principale qui lie le tout. Elle lit du code Psil d'un fichier et l'évalue.

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

```
% ghci
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
ghci> :load "psil.hs"
[1 of 2] Compiling Main             ( psil.hs, interpreted )

psil.hs:317:1: warning: [GHC-62161] [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘eval’:
        Patterns of type ‘Env’, ‘Lexp’ not matched:
            _ (Lvar _)
            _ (Labs _ _)
            _ (Lapply _ _)
            _ (Lnew _ _)
            ...
    |
317 | eval _ (Lnum n) = Vnum n
    | ^^^^^^^^^^^^^^^^^^^^^^^^
Ok, one module loaded.
ghci> run "e2025-exemples.psil"
[2,*** Exception: e2025.hs:317:1-24: Non-exhaustive patterns ...

ghci> :quit
Leaving GHCi.
%
```

Une fois complété, votre code ne devrait plus souffrir d'avertissement comme celui ci-dessus.

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.psil`, similaire à `exemples.psil`, mais qui contient au moins 5 tests que *vous* avez écrits (avec en commentaire une description de ce qui est testé ainsi que 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) et exercer différentes parties de l'implémentation, pour détecter des erreurs différentes.
- Votre implantation de Psil 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 les `exemples.psil` et les `test.psil` 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 les fichiers suivants: `psil.hs`, `tests.psil`, et `rapport.tex`. Il sont à remettre 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*, 20% pour le rapport, 20% 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 inefficace.