# Travail pratique #1

IFT-2035

18 octobre 2021

¡¡ Dû le 21 octobre à 23h59 ! !

## 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.

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Des groupes de 3 ou plus sont **exclus**.

| | | |
|---|---|---|
| $e$ | ::= $n$ | Un entier signé en décimal |
| | $\mid x$ | Référence à une variable |
| | $\mid (e_0\ e_1\ ...\ e_n)$ | Un appel de fonction |
| | $\mid$ (lambda $(x_1\ ...\ x_n)\ e$) | Une fonction de $n$ arguments |
| | $\mid$ (dlet $(d_1\ ...\ d_n)\ e$) | Déclarations locales dynamiques |
| | $\mid$ (slet $(d_1\ ...\ d_n)\ e$) | Déclarations locales statiques |
| | $\mid + \mid - \mid * \mid /$ | Opérations arithmétiques prédéfinies |
| | $\mid \leq \mid < \mid \geq \mid >\mid =$ | Comparaisons arithmétiques prédéfinies |
| | $\mid$ true $\mid$ false | Valeurs booléennes prédéfinies |
| | $\mid$ (if $e_1\ e_2\ e_3$) | Opération conditionnelle |
| | $\mid$ (cons $tag\ e_1\ ...\ e_n$) | Construction de structure |
| | $\mid$ (case $e\ b_1\ ...\ b_n$) | Filtrage sur les listes |
| $b$ | ::= $((tag\ x_1\ ...\ x_n)\ e)$ | Branche de filtrage |
| | $\mid$ (_ $e$) | Branche par défaut |
| $d$ | ::= $(x\ e)$ | 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. Tout comme Lisp, et au contraire de Haskell, c'est un langage typé dynamiquement.

Slip a 3 types de données : les entiers, Les fonctions, et les *structures* qui peuvent comporter un nombre arbitraire de champs et viennent avec un *tag* qui permet de les distinguer. Les booléens ne sont pas un type spécial : les valeurs booléennes prédéfinies true et false ne sont en fait que des structures avec 0 champs et avec comme tag true et false respectivement.

La fonction prédéfinie cons construit une structure, et on peut tester le tag d'une structure ainsi qu'en extraire ses champs, avec l'opération case, qui fonctionne un peu comme celle de Haskell.

Les formes slet et dlet sont utilisées pour donner des noms à des définitions locales. Il y en a deux, car Slip, tout comme Lisp, offre autant la portée dynamique que la portée statique : les variables définies avec slet obéissent aux règles de la portée statique/lexicale, alors que celles définies avec dlet utilisent la portée dynamique. Exemple :

$$\begin{array}{l}(\text{slet } ((x\ 2)\\ \qquad\ (y\ 3))\\ \ (+\ x\ y))\end{array} \quad \leadsto^* \quad 5$$

Vu que beaucoup de définitions locales sont des fonctions, Slip offre une syntaxe particulière pour définir des fonctions :

$$\begin{array}{l}(\mathsf{slet}\ ((y\ 10)\\ \qquad((\mathit{div2}\ x)\\ \qquad\ (+\ (/\ x\ 2)\ y)))\\ \ (\mathit{div2}\ y))\end{array} \quad \rightsquigarrow^* \quad 15$$

Les définitions avec slet ne sont pas récursives, mais séquentielles :

$$\begin{array}{l}(\mathsf{slet}\ (((f\ x)\ (+\ x\ 5))\\ \qquad((f\ x)\ (f\ (*\ x\ 2))))\\ \ (f\ 3))\end{array} \quad \rightsquigarrow^* \quad 11$$

La syntaxe de dlet et de slet sont identiques, et leur comportement est le même mis à part en ce qui concerne la portée.

## 2.1 Sucre syntaxique

La syntaxe donnée plus haut inclus du sucre syntaxique : Plus précisément, les équivalences suivantes sont vraies pour les expressions :

$$\begin{array}{rcl}(\mathsf{if}\ e_1\ e_2\ e_3) & \Longleftrightarrow & (\mathsf{case}\ e_1\ ((\mathsf{true})\ e_2)\ ((\mathsf{false})\ e_3))\\ (\mathsf{slet}\ (d_1\ ...\ d_n)\ e) & \Longleftrightarrow & (\mathsf{slet}\ (d_1)\ (\mathsf{slet}\ (...\ d_n)\ e))\\ (\mathsf{dlet}\ (d_1\ ...\ d_n)\ e) & \Longleftrightarrow & (\mathsf{dlet}\ (d_1)\ (\mathsf{dlet}\ (...\ d_n)\ e))\\ (e_0\ e_1\ e_2\ ...\ e_n) & \Longleftrightarrow & (..((e_0\ e_1)\ e_2)\ ...\ e_n)\\ (\mathsf{lambda}\ (x_1\ ...\ x_n)\ e) & \Longleftrightarrow & (\mathsf{lambda}\ (x_1)\ ...\ (\mathsf{lambda}\ (x_n)\ e)..)\end{array}$$

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 :

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

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 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 structures (dénotées $[tag\ v_1\ ...\ v_n]$). Les variables introduites par dlet utilisent la portée dynamique, alors que toutes les autres utilisent la portée statique.

Les règles d'évaluation fondamentales sont les suivantes :

$$\begin{array}{rcl} (v\ (\mathsf{lambda}\ (x)\ e)) & \rightsquigarrow & e[v/x] \\ (\mathsf{slet}\ ((x\ v))\ e) & \rightsquigarrow & e[v/x] \end{array}$$

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.

La syntaxe d'appel de fonction place la fonction en seconde position, un peu comme les méthodes dans les langages orienté objets, de sorte que cela ressemble un peu à un pipeline : $(x\ f_1\ f_2\ ...)$ correspond à passer $x$ à $f_1$ puis à passer le résultat à $f_2$ etc...

En plus des deux règles ci-dessus (et des primitives arithmétiques), les *structures* se comportent comme suit :

$$\begin{array}{l} (\mathsf{cons}\ tag\ v_1\ ...\ v_n) \rightsquigarrow [tag\ v_1\ ...\ v_n] \\ (\mathsf{case}\ v\ (\_\ e)) \rightsquigarrow e \\ (\mathsf{case}\ v\ ((tag\ x_1\ ...\ x_n)\ e)\ ...\ b_n) \\ \qquad \rightsquigarrow \begin{cases} e[v_1, ..., v_n/x_1, ..., x_n] & \text{si } v = [tag\ v_1\ ...\ v_n] \\ (\mathsf{case}\ v\ ...\ b_n) & \text{sinon} \end{cases} \end{array}$$

# 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. C'est une sorte d'arbre de syntaxe abstraite générique (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 autre 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 : *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 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 (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, +, -, false, ... sont simplement des variables prédéfinies, et le sucre syntaxique n'est plus disponible, donc il n'y a plus de if et les let ne peuvent définir qu'une variable à la fois.

Elle est définie par le type :

```
data Lexp = Lnum Int
          | Lvar Var
          | Lfn Var Lexp
          | Lpipe Lexp Lexp
          | Lcons Tag [Lexp]
          | Lcase Lexp [(Pat, Lexp)]
          | Llet BindingType 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.

## 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
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> :l "slip.hs"
[1 of 1] Compiling Main             ( a2021.hs, interpreted )
Ok, one module loaded.
*Main> run "exemples.slip"
[2,*** Exception: Can't eval: Lvar "+"
CallStack (from HasCallStack):
  error, called at slip.hs:255:14 in main:Main
*Main>
```

Lorsque votre travail sera fini, ce test devrait renvoyer plus de valeurs que juste le "2" ci-dessus et terminer sans erreurs.

# 5 À faire

Vous allez devoir compléter l'implantation de ce langage, c'est à dire principalement compléter *s2l* 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 : 30% pour *s2l*, 25% pour le rapport, 15% pour vos tests, et 30% pour *eval*.
— 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, 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.