

# Travail pratique #1

IFT-2035

October 6, 2009

ii Dû le 28 octobre à 8h30 !!

## 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'occurrence 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 4 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 PDF exclusivement) 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. Si un étudiant ne trouve pas de partenaire, il doit me contacter au plus tard lundi 13 octobre **avant** le cours. Des groupes de 3 ou plus sont **exclus**.

## 2 Une sorte de Lisp

Vous allez travailler sur l'implantation d'une variante du langage Lisp. Ce langage comprendra des expressions de la forme suivante:

$e ::= n$	Un entier signé en décimal
$x$	Une variable
$(e_0 e_1 \dots e_n)$	Un appel de fonction; les arguments sont <i>curried</i>
$(\text{lambda } (x_1 \dots x_n) e)$	Une fonction; les arguments sont <i>curried</i>
$(\text{let } (d_1 \dots d_n) e)$	Ajout de déclarations locales
$+ \mid - \mid * \mid /$	Opérations arithmétiques prédéfinies
$\text{cons} \mid \text{nil} \mid \text{car} \mid \text{cdr}$	Fonctions prédéfinies de manipulation de listes
$(\text{ifnil } e e_n e_c)$	Évalue $e_n$ si $e$ est <i>nil</i> et $e_c$ sinon
$d ::= (x e)$	
$((x x_1 \dots x_n) e)$	

### 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 equivalences suivantes sont vraies:

$$\begin{aligned} (e_0 e_1 e_2 \dots e_n) &\iff \dots((e_0 e_1) e_2) \dots e_n \\ (\text{lambda } (x_1 \dots x_n) e) &\iff (\text{lambda } (x_1) \dots (\text{lambda } (x_n) e) \dots) \end{aligned}$$

La fonction prédéfinie **cons** construit une paire (comme le  $(,)$  de Haskell), **car** en extrait la valeur de gauche (comme **fst**) et **cdr** en extrait la valeur de droite (comme **snd**). **nil** est une constante prédéfinie qui est utilisée pour représenter la liste vide. **cons** est aussi utilisé pour créer des listes non vides; ainsi, si l'on fait abstraction des différences de typage, **cons** est comme l'opérateur  $(:)$  de Haskell, **nil** est comme  $[]$ , **car** est comme **head**, et **cdr** est comme **tail**.

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

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

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

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

Cette syntaxe se réduit à la syntaxe plus rudimentaire avec l'équivalence suivante:

$$(\text{let } (((x x_1 \dots x_n) e)) e) \iff (\text{let } ((x (\text{lambda } (x_1 \dots x_n) e))) e)$$

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

$$\begin{aligned} (\text{let } (((\text{even } x) (\text{ifnil } x 1 (\text{odd } (\text{cdr } x)))) \\ ((\text{odd } x) (\text{ifnil } x 0 (\text{even } (\text{cdr } x)))))) \\ (\text{odd } (\text{cons } 0 (\text{cons } 0 (\text{cons } 0 \text{ nil})))))) \rightsquigarrow^* 1 \end{aligned}$$

## 2.2 Sémantique dynamique

Les valeurs manipulées sont les entiers, les fonctions, la valeur nil notée  $\square$ , et les paires notées  $[v_1 . v_2]$ . De plus la notation de paires 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 “ $\square$ ” final peut s'éliminer, de sorte que  $[v_1 v_2]$  est équivalent à  $[v_1 . [v_2 . \square]]$ .

Les règles d'évaluation fondamentales sont les suivantes:

$$\begin{aligned} ((\text{lambda } (x) e) v) &\rightsquigarrow e[v/x] \\ (\text{let } ((x_1 v_1)\dots(x_n v_n)) e) &\rightsquigarrow e[v_1, \dots, v_n/x_1, \dots, x_n] \end{aligned}$$

où la notation  $e[v/x]$  représente l'expression  $e$  dans un environnement où la variable  $x$  prend la valeur  $v$ .

En plus des deux règles  $\beta$  ci-dessus, les différentes constantes prédéfinies se comportent comme suit:

$$\begin{aligned} (+ n_1 n_2) &\rightsquigarrow n_1 + n_2 \\ (- n_1 n_2) &\rightsquigarrow n_1 - n_2 \\ (* n_1 n_2) &\rightsquigarrow n_1 \times n_2 \\ (/ n_1 n_2) &\rightsquigarrow n_1 \div n_2 \\ \text{nil} &\rightsquigarrow \square \\ (\text{cons } v_1 v_2) &\rightsquigarrow [v_1 . v_2] \\ (\text{car } [v_1 . v_2]) &\rightsquigarrow v_1 \\ (\text{cdr } [v_1 . v_2]) &\rightsquigarrow v_2 \\ (\text{ifnil } \square e_n e_c) &\rightsquigarrow e_n \\ (\text{ifnil } [v_1 . v_2] e_n e_c) &\rightsquigarrow e_c \end{aligned}$$

Donc il s'agit d'une variante du  $\lambda$ -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. Ce n'est pas encore un arbre de syntaxe abstraite.
2. Une deuxième phase, appelée *compile*, termine l'analyse syntaxique et commence la compilation, en transformant cet arbre en un vrai arbre de syntaxe abstraite dans la représentation appelée  $L$  (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  $\dots \iff \dots$ ), et doit faire quelques ajustement supplémentaire car  $Lexp$  utilise une forme d'appel de fonction un peu plus restrictive, et appelle ses primitives de manière non *curried*.

3. Une troisième phase, appelée *optimize*, est censée rendre le code plus compact et plus efficace en appliquant différentes règles de simplification.
4. Finalement, une fonction *eval* procède à l'évaluation de l'expression par interprétation.

Une partie de l'implantation est déjà fournie: la première et la dernière phase ainsi que le début de la deuxième. Votre travail consistera à compléter *compile*, puis dans un deuxième temps à développer la fonction *optimize*.

### 3.1 Analyse lexicale et syntaxique

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

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

$n$  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 head tail` et du marqueur de début `Snil`. *tail* est le dernier élément de la liste et *head* est le reste de la liste.

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. Elle est définie par le type:

```
data Lexp = Lnum Int
          | Lvar Var
```

```

| Llambda [Var] Lexp
| Lapp Var [Lexp]
| Lif Lexp Lexp Lexp
| Llet Var Lexp Lexp
| Lfix [(Var,Lexp)] Lexp

```

Dans cette représentation, `cons`, `+`, `...` sont simplement des variables prédéfinies. L'expression conditionnelle `ifnil` est remplacée par l'expression conditionnelle générique `Lif` et par une primitive `ifnil` qui renvoie un booléen. Le `Llet` permet de définir une variable locale avec une valeur particulière; et le `Lfix` est comme le `Llet` sauf qu'il permet de définir plusieurs variables et qu'il autorise la récursion (y compris la récursion mutuelle). La fonction *compile* doit transformer un code source tel que:

```

(let (((f1 x) 1)
      ((f2 x) (+ x 1))
      ...))

```

en une forme telle que:

```

Lfix [{"f1", Llambda ["x"] (Lnum 1)},
      {"f2", Llambda ["x"] (Lapp "+" [Lvar "x", Lnum 1])}]
...

```

Cette phase est plus complexe qu'elle n'y paraît si on veut vraiment qu'elle traite tous les programmes correctement. Exemple de cas non-trivial:

```

(cons (+ 1)
      (let ((+ x y z) (ifnil z y x))
        (+ 1)))

```

De plus, *compile* va devoir parfois introduire de nouvelles variables, donc il faudra faire attention à éviter les conflits de noms.

## 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 définit 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*.

*compile* 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*.

*main* et *test* sont les fonctions principales qui lient le tout. En première approximation, elles correspondent au départ à

```
eval env0 (compile (readSexp <code>))
```

Voilà ci-dessous un exemple de session interactive sur une machine GNU/Linux, lorsque le code est complété:

```
% hugs sol.hs
-- -- -- -- ---- --
||  || ||  || ||  ||  ||__  Hugs 98: Based on the Haskell 98 standard
||__||  ||__||  ||__||  __||  Copyright (c) 1994-2003
||---||          ___||      World Wide Web: http://haskell.org/hugs
||  ||          Report bugs to: hugs-bugs@haskell.org
||  || Version: November 2003 -----

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Main> test "1"
1
Main> main "exemples.slip"
[5,8,-6,1,3,3,1,[[6 7 8 9] . 10]]
Main> [Leaving Hugs]
%
```

## 5 À faire

Vous allez devoir compléter l'implantation de ce langage comme suit.

### 5.1 Étape 1

La fonction *compile* fournie ne reconnaît que la syntaxe la plus basique. Elle ne reconnaît et ne réécrit pas le sucre syntaxique. De plus elle ne fonctionne que pour les *let* non récursifs et qui ne définissent qu'une variable. De même *eval* ne sait pas gérer les définitions récursives.

Dans la première étape vous allez donc devoir corriger ces 2 problèmes:

1. Étendre *compile* de manière à gérer le sucre syntaxique, tel que décrit à la section ??.

2. Étendre *compile* et *eval* de sorte à accepter les *let* qui définissent plus d'une variable à la fois et/ou qui utilisent la récursion. Vous allez pour cela devoir utiliser *Lfix* au lieu de *Llet*.

### 5.1.1 Étape 2

Tout compilateur qui se respecte inclut une ou plusieurs phases d'optimisation qui tentent de rendre le code plus efficace, c'est à dire, soit plus compact, ou plus rapide, ou plus frugal (dans sa consommation de mémoire), ou une combinaison.

La fonction *optimize* fournie ne fait presque rien d'intéressant, donc vous allez devoir l'écrire plus ou moins dans sa totalité. Les optimisations que l'on peut vouloir appliquer à notre langage ne manquent pas, mais nous allons nous limiter à des optimisations simples et courantes, qui tentent simplement d'éliminer les formes visiblement inefficaces. Plus précisément, vous devriez pouvoir optimiser des expressions comme suit:

$(\text{let } ((x\ 3)) \dots x \dots x \dots)$	$\rightarrow (\text{let } ((x\ 3)) \dots 3 \dots 3 \dots)$	propagation de constantes
$((\text{lambda } (x) e_1) e_2)$	$\rightarrow (\text{let } ((x\ e_2)) e_1)$	sorte de $\beta$ -réduction
$(\text{let } (((f\ x) (g\ x))) \dots f \dots f \dots)$	$\rightarrow (\text{let } (((f\ x) (g\ x))) \dots g \dots g \dots)$	$\eta$ -réduction
$(\text{let } ((x\ v)) e)$	$\rightarrow e$	élimination de code mort
$(\text{let } ((x (\text{let } d\ e_1))) e_2)$	$\rightarrow (\text{let } d (\text{let } ((x\ e_1)) e_2))$	associativité de <i>let</i>
$(\text{car } (\text{cons } v_1\ v_2))$	$\rightarrow v_1$	$\beta$ -réduction sur <i>car</i>
$(\text{cdr } (\text{cons } v_1\ v_2))$	$\rightarrow v_2$	$\beta$ -réduction sur <i>car</i>

La *propagation de constantes* peut s'appliquer non seulement aux constantes entières comme dans l'exemple, mais à n'importe quelle expression. Si on l'applique aux définitions de fonctions, combinée avec la deuxième règle ci-dessus, on obtient le *inlining* ("copie en ligne") qui remplace chaque appel à la fonction par une copie du code de la fonction. Votre *optimize* devrait être capable de faire de tel *inlining*. Cependant, trop de copies de code mènent facilement à une explosion de la taille du code, donc la propagation de constantes doit être contrôlée. Plus précisément, vous devez vous assurer que la propagation de constante n'augmente pas la taille du code.

La *réduction  $\eta$*  et la propagation de constantes vont main dans la main avec l'*élimination de code mort*, vu que les fonctions  $\eta$ -réduites et les constantes propagées se retrouvent normalement candidates à l'élimination.

La règle d'*associativité de let* n'est pas directement une règle d'optimisation, mais elle permet parfois d'exposer des opportunités d'application d'autres règles.

Par exemple:

```
(let ((x (let ((y (+ 1 z))) (cons y y)))) (car x))
  ↓ associativité de let
(let ((y (+ 1 z))) (let ((x (cons y y))) (car x)))
  ↓ propagation de constante
(let ((y (+ 1 z))) (car (cons y y)))
  ↓ β-réduction sur car
(let ((y (+ 1 z))) y)
  ↓ propagation de constante
(+ 1 z)
```

Ces optimisations sont exprimées dans la syntaxe du code source, mais vous allez devoir les implanter dans la forme *Lexp*, ce qui est à vrai dire souvent plus facile. Il y a aussi des optimisations que l'on peut faire dans *Lexp* qui n'ont pas d'équivalent dans le code source: par exemple, remplacer un `Lfix` par un `Llet`, ou remplacer un appel *curried* par un appel non-*curried*.

Faites attention au fait que ces règles ne sont pas universellement valides. C'est à vous de déterminer quand une règle est valide et quand elle ne l'est pas. De plus, comme dans l'exemple précédent, il arrive souvent que l'application d'une règle dépende d'une autre, et donc l'ordre d'application de ces règles peut être important (quoiqu'il n'y ait pas d'ordre optimal).

Vu que l'implantation fournie de *optimize* génère déjà du code correct, la notation de cette étape va tenir compte non seulement du fait que votre code génère des programmes corrects mais aussi qu'il optimise bien. Donc le code généré doit être aussi compact que possible (tout en restant correct, bien sûr).

## 5.2 Remise

Pour la remise, due le 28 octobre à 8h30 précise, vous devez remettre deux fichiers:

```
% remise ift2035 tp1 slip.hs rapport.pdf
```

## 6 Détails

- La note sera divisée comme suit: 50% pour *compile*, 25% pour le rapport et 25% pour *optimize*.
- 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) et souvent incompréhensible. L'efficacité de votre code est sans importance, sauf s'il utilise un algorithme vraiment particulièrement ridiculement inefficace.