Travail pratique #1

IFT-2035
Nathan Razafindrakoto 20254813
Yasmine Ben Youssef 20237210

13 octobre 2024

1 Initialisation du projet

Yasmine : "C'est quoi GitHub ?"
Nathan : "C'est quoi Haskell ?"
Après avoir passé 30 minutes à chercher le meilleur template à appliquer au rapport LaTeX, on a plutôt décidé de réutiliser l'énoncé du TP1.

2 Prise en main

On a commencé par tester la fonction *readSexp* pendant 30 minutes afin de comprendre à quoi ressemble une expression telle que *(let x 2 (let y 3 (+ x y))* après être passée par l'analyseur syntaxique.

La partie la plus dure pour implémenter *s2l* a été de comprendre à quoi la fonction sert. "Elle élimine le sucre syntaxique" quel sucre syntaxique ? Celle de la déclaration de fonction ? Est-ce qu'il y a d'autres sucres syntaxiques ?

Une fois que ça a été clarifié, on a travaillé sur toutes les différentes formes de Sexp possibles selon la syntaxe de Slip afin de les traiter une par une ensuite dans la fonction *s2l*.

Dans la phase d'implémentation, on devait plusieurs fois se reprendre pour comprendre que *s2l* ne sert qu'à éliminer le sucre syntaxique de la déclaration de fonction et transformer le code dans un format plus facile à manipuler pour la prochaine fonction à implémenter.

Sauf que pour comprendre sous quel format le code doit être retourné, on a décidé d'aller voir la fonction *eval*.

3 *eval*

La partie la plus coûteuse en temps était de penser à tous les cas possibles pour éviter que des expressions valides soient considérées invalides (donc non traitées) par l'interpréteur.
Les cas d'*eval* pour *Llet*, *Lvar*, *Ltest* étaient faciles à faire grâce aux séances de

démonstrations et les exercices faits en classe. Pour *Lfob*, on a écrit une ligne simple en espérant que ça marche. Jusque là, cette ligne a marché sans accroc. À ce rythme, on pensait que *eval* serait facile à implémenter finalement.

Haha, finalement non, ce n'était pas du tout facile. On a mis du temps à savoir comment appeler une fonction anonyme de sorte à ce que ses arguments soient déjà évalués et ajoutés dans l'environnement avant de l'évaluer.

RÉVÉLATION DE FOU : Après avoir essayé de créer l'évaluation pour l'appel de fonction et la construction de fonctions anonymes (que Nathan a apparemment réussi du premier coup), on s'est rendus compte que

$$\begin{aligned}\mathsf{Lsend}\ (\mathsf{Lfob}\ [\text{"}x\text{"}]\ (\mathsf{Lsend}\ (\mathsf{Lfob}\ [\text{"}y\text{"}]\ (\mathsf{Lsend}\ (\mathsf{Lvar}\ \text{"}*\text{"})\\ [\mathsf{Lvar}\ \text{"}x\text{"}, \mathsf{Lvar}\ \text{"}y\text{"}]))\ [\mathsf{Lnum}\ 5]))[\mathsf{Lnum}\ 3]\end{aligned}$$

équivaut (dans un sens informel) à

$$\mathsf{Lsend}\ (\mathsf{Lfob}\ [\text{"}x\text{"}, \text{"}y\text{"}]\ (\mathsf{Lsend}\ (\mathsf{Lvar}\ \text{"}*\text{"})\ [\mathsf{Lvar}\ \text{"}x\text{"}, \mathsf{Lvar}\ \text{"}y\text{"}]))\ [\mathsf{Lnum}\ 5, \mathsf{Lnum}\ 3]$$

Ça nous a drastiquement aidés à comprendre sous quelle forme les fonctions anonymes doivent être traduites à travers *s2l*. Il nous reste alors à voir la forme générale et la traduire. yess...

4 Retour sur *s2l*

C'était extrêmement pénible de déterminer la meilleure façon de transformer les imbrications de constructeurs de fonctions comme *(((fob (x) (fob (y) (* x y))) 3) 5)* à travers *s2l*. On a d'abord pensé à tout simplement imbriquer des *Lfob* dans l'expression Lexp finale, mais ça aurait été un calvaire à gérer ensuite dans la fonction *eval*, surtout que Nathan n'avait pas envie de changer sa version actuelle car il en était fier et qu'elle réussissait bizarrement à marcher même sur des imbrications de fonctions prédéterminées.
Notre solution finale était alors de faire en sorte que toute forme de création imbriquée de fonction rassemble toutes les variables arguments de la fonction finale dans une seule liste, ainsi que les valeurs assignées à ces arguments dans l'appel de la fonction dans une autre liste. Par exemple :

$$\mathsf{s2l}\ (\mathsf{readSexp}\ \text{"}(((\mathsf{fob}\ (x)\ \mathsf{fob}\ (y)\ (*\ x\ y)))\ 3)\ 5\text{"})$$

devrait donner

$$\mathsf{Lsend}\ (\mathsf{Lfob}\ [\text{"}x\text{"}, \text{"}y\text{"}]\ (\mathsf{Lsend}\ (\mathsf{Lvar}\ \text{"}*\text{"}\ [\mathsf{Lvar}\ \text{"}x\text{"}, \mathsf{Lvar}\ \text{"}y\text{"}]))\ [\mathsf{Lnum}\ 3, \mathsf{Lnum}\ 5]$$

C'était bien plus compliqué à faire mais une fois qu'on l'a fini, la fonction *eval* a immédiatement marché. C'était aussi drôle de voir que le typage dynamique n'a posé aucun problème car l'évaluation s'en chargeait toute seule.
Concernant la déclaration de variables, l'expression conditionnelle, l'appel de

fonction et la déclaration locale non récursive, leur transformation était triviale une fois que celle de *fob* a été implémentée.
Il est alors venu le temps d'affronter la bête noire qui nous terrifiait dès la première lecture du TP...

5 fix

5.1 s2l

AAAAAAAAAAAAAAAAAAAAAAA
...what ?
Au final c'est plus simple que prévu, il suffit d'extraire les déclarations fournies puis de les ajouter à l'environnement avant d'évaluer l'expression donnée en second argument.

C'EST CE QU'ON S'EST DITS JUSQU'À CE QU'ON SE RENDE COMPTE QUE LES DÉCLARATIONS DE LA LISTE PEUVENT RÉFÉRENCER RÉCURSIVEMENT DES DÉCLARATIONS DE LA MÊME LISTE.

Haha ça concerne la fonction *eval*, on n'a pas à s'en inquiéter pour l'instant.
...On a quand même dû définir 3 fonctions auxiliaires pour extraire correctement les déclarations en distinguant celles de variables simples et celles de fonctions. Le fait que la représentation syntaxique impose que la première déclaration soit le premier argument du Snode et que toutes les autres déclarations soient dans la liste donnée en deuxième argument du Snode a vraiment compliqué la tâche. À la fois pour *fix* et pour *fob* quand on y pense.

5.2 eval

Bilan : 4 heures de réflexion pour une fonction qui fait techniquement 4 lignes. Heureusement que tous les autres cas de *eval* étaient suffisamment robustes pour tenir les 74 tests avec imbrication préalables qu'on a faits, sinon on sent que ça aurait plutôt été 40 heures de réflexion...

Le coeur du problème était de savoir comment évaluer récursivement chaque variable sans créer de conflits avec les autres qui n'ont pas encore été déclarés au moment de l'évaluation, par exemple si une variable non déclarée est appelée pour déclarer la première variable etc...
Nathan : "Il faut découper les enfants en deux récursivement"
Notre solution était la suivante :

- On commence par déclarer les bindings (les variables et fonctions).
- Ensuite, chaque binding est immédiatement évalué en fonction de l'environnement courant, qui contient tous les autres bindings déclarés, même si certains ne sont pas encore entièrement évalués.

— Les valeurs des variables sont résolues au fur et à mesure, mais les variables peuvent se référer les unes aux autres, car elles sont toutes présentes dans l'environnement dès le début.

Le soupir de soulagement de Nathan au moment de voir que run "exemples.slip" fait passer tous les tests aurait été suffisant pour renverser les maisons des trois petits cochons en même temps (même celle en briques).

6 Conclusion

Yasmine a déjà peur du TP2