Description de notre experience pour les points suivants : Problèmes, surprises, choix, options rejetées, etc.

1 Parfaire sa connaissance de Haskell

Haskell n'est pas un langage très intuitif à prime abord. Penser et déconstruire un problème par récursions semble être un défi de taille pour ceux qui n'y ont pas l'habitude. Mais à travers les différents problèmes rencontrés, notamment dans ce TP et les devoirs, en élargissant pas à pas notre sphère d'aisance, nous sommes arrivés à avoir une bonne prise en main de la logique du langage. Tel que mentionné dans le cours, il est possible (et souhaitable) de considérer un code comme une preuve mathématique, en établissant systématiquement le chemin des déductions qui partent du ou des types d'entrée jusqu'au type de sortie, et Haskell s'y prête grandement. En ayant en tête l'idée de preuve mathématique (notamment par induction), la pensée par récursion devient plus simple : on débute par le cas de base, puis par un cas n et $n+1$ qui représentent les autres cas de figures. Penser de telle manière est pratique pour bien comprendre le fonctionnement des fonctions et pour anticiper les erreurs. Nous avons donc essayé de suivre cette démarche logique et systématique dans notre TP, bien que ce ne soit pas un muscle à l'usage familier.

2 Lire et comprendre la donnée

Pour bien comprendre l'énoncé du TP, nous avons décidé de le lire chacun de notre côté puis de se rencontrer pour résumer les différentes parties. En déconstruisant le travail de la sorte, nous avons mieux compris les différentes tâches que nous avions à faire. [développer]. Nous nous sommes rendu compte, en cours de route, que nous avions interprété certaine partie d'une mauvaise manière et que nous n'étions, au final, peut-être pas vraiment certain de ce que nous faisions.

3 Lire, trouver et comprendre les parties importantes du code fourni

Pour ce qui est du code donné, la longueur du fichier nous a d'abord parut inquiétante et nous avons surévalué la tâche à faire. Mais en ayant bien défini les sections à compléter grâce à notre synthèse de la donnée, nous avons simplement considéré de gros bouts du code comme des boites noires pour nous concentrer exclusivement sur *s2l* et *eval*. [développer] Avoir travailler sur la construction d'évaluateurs dans les différents devoirs nous a permis de comprendre ce que nous devions faire dans *eval* assez rapidement, mais nous avions quelques doutes sur la manière dont fonctionnait *s2l* et comment était défini la structure des *sexp* et des *lexp*. Nous avons débuté par lister les cas que nous devions effectuer. Pour

$s2l$, il était trivial que nous avions à transformer les expressions $Snil$, $Ssym$, $Snum$ et $Snode$ en $Lnum$, $Lbool$, $Lvar$, $Ltest$, $Lfob$, $Lsend$, $Llet$ et $Lfix$. Puis pour $eval$, il fallait prendre ces derniers cas et les transformer en $Vnum$, $Vbool$, $Vbuiltin$ et $Vfob$. Avant de commencer à coder, nous voulions comprendre la logique des transformations (et particulièrement $s2l$). Les cas les plus simples étaient ceux $Snum$ et des $Ssym$ que nous comprenions assez bien, mais pour les autres, il nous semblait très difficile de trouver un point de départ. Que doit faire $s2l$ au juste ? Comment sera t-il possible d'interpréter un $sexp$ en $lexp$ pour des booléens, des fonctions, etc. ? Nous pensions déjà en termes sémantiques, sur la valeur des arguments, des opérations, etc. Nous faisions fausse route et cela nous a conduit à considérer le travail comme beaucoup plus difficile qu'il ne l'était. Lorsque nous avions bien intériorisé le fait que $s2l$ manipule tout simplement des arbres syntaxique, le problème nous est apparu plus simple. Nous comprenions qu'il n'était que question de quelle disposition de symboles $Sexp$ donne telle disposition de symboles $Lexp$.

4 Compléter le code fourni

Ainsi le problème bien opérationnalisé, nous avons simplement commencer par les cas de figure qui nous semblaient simples, à savoir, les booléens, qui sont syntaxiquement simples (avec deux choix : "$true$" ou "$false$") puis ceux qui débutaient par un symbole particulier ("if", "fob", "let", "fix"). if nous a semblé relativement trivial ; nous avons trois expressions, une pour l'énoncé conditionnel, une pour la condition "true" et une pour la condition "false" et nous retournons un ltest avec chacune des expressions transformées en $Lexp$. Pour let, la transformation nous semblait aussi assez simple ; on prend une assignation et un corps d'expression, puis nous retournons tout simplement un $Llet$ avec la variable assignée, la transformation de la valeur assignée et la transformation du corps de l'expression. La logique nous semblait similaire pour $Lfix$ mais de manière récursive sur un ensemble indéterminé d'assignations. Notre approche était de d'établir la logique des fonctions $s2l$ puis celle de $eval$ avant de tester. Notre méthode nous semblait bonne, mais nous nous sommes rendu compte en cours de route de son inéficacité. En relisant bien la données [Cas de Fob plus difficile].

[sucre syntaxique] -¿ on aurait du le faire au début, on s'Est rendu compte en le lisant.

[à développer]. Problème avec fob, et fix... Problème avec lsend...

Nous pensions qu'une approche en largeur plutôt qu'en profondeur était une bonne stratégie telle que suggérée dans la donnée.

intuition, test, vérification et recalibrage !