Devoir 2
Concept des langages de programmation

Vincent Antaki (p1038646)
Émile Trottier (p1054384)

8 décembre 2014

Résumé

Implémentation d'un splay-tree en Scheme

1 Fonctionnement général du programme

La fonction *traiter* de notre programme a le comportement typique d'une fonction main(). La compréhension de cette méthode donne une bonne idée générale de notre programme. Celle-ci est appelé récursivement pour permettre autant de requête que l'usager souhaite.

La première étape est de reconnaître quel est le type de requête l'usager demande. Pour cela, nous utilisons d'abord la fonction *eval-expr* sur l'expression de l'usager qui nous la met sous une bonne forme. Ainsi, si la requête contient un symbole '=' et qu'il n'y a aucun caractères après le 'key=', il s'agit d'une demande de suppression de mot et notre fonction la met sous la forme '(- key). Lors d'une entrée de la forme 'key=definition', la liste '(= key definition) est retournée parce qu'il s'agit de l'ajout d'un mot au dictionnaire. Sinon, si le caractère '=' est tout simplement absent de la chaîne, c'est une recherche d'un mot et la forme '(% key) est utilisée. Cette manière de faire nous permet, une fois de retour dans la fonction *traiter* de différencier les cas d'utilisation et d'accéder facilement la clé et la définition lorsque qu'applicable.

Ainsi, notre fonction *traiter* n'a qu'à vérifier le premier élément de la liste retournée par *eval-expr* pour savoir quel cas elle doit gérer. Dans le cas d'une suppression, nous appelons la fonction *node-remove* qui l'exécute. Dans le cas d'un ajout au dictionnaire par concaténation de terme, on transforme la définition de sorte qu'elle soit une liste de caractères (avec mémoire partagée bien sûr). Nous avons qu'à ajouter un nouveau noeud à l'arbre par la suite. Dans le dernier cas où l'usager demande de rechercher un mot, celui-ci est recherché par la fonction *node-find* de manière récursive dans notre arbre.

Par souci de simplicité et pour garder un niveau d'abstraction assez élevé dans cette brève explication du programme, nous avons omis plusieurs détails, comme le traitement des erreurs, le splay et la conservation des données. Il va sans dire qu'ils sont néanmoins pris en charge dans le programme, comme nous l'expliquerons dans les prochains paragraphes.

2 Résolution de problème de programmation

2.1 Analyse syntaxique d'une requête et lecture d'une requête de longueur arbitraire

Tel que mentionné plus tôt, nous avons la fonction *eval-expr* qui nous transforme la requête sous une forme que nous jugeons plus repésentative et concise. Si le test d'appartenance au caractère '=' dans la requête réussi et que la partie qui suit le '=' est vide, alors notre fonction reconnaît une suppression et retourne une forme appropriée. Si la partie après le '=' est non-vide, alors il s'agit de l'ajout d'un mot au dictionnaire. Deux situations sont alors possibles : soit la suite contient le caractère '+' et il faut faire une concaténation de définitions, soit il n'y apparaît pas et il s'agit d'une définition bien normale. Si la clé du nouveau noeud existe déjà dans le dictionnaire, la fonction *node-insert* se charge de remplacer le noeud existant. Sinon, c'est-à-dire si le signe '=' n'apparait pas dans la requête, alors la conclusion de notre programme est que l'usager veut rechercher la définition d'un certain terme. Il est à noter que dès qu'une concaténation contient un seul mot inexistant, l'opération est annulée et le message "terme inconnu" est affiché et que peu importe la requête, son type est affiché à l'écran.

En ce qui trait à la lecture de la demande de l'usager, la fonction *go* donnée s'en charge.

Après chaque fin de requête de l'usager, c'est-à-dire à chaque fois qu'il tape la touche "enter", la ligne nous est passée sous forme de liste de caractère. Celle-ci peut donc être de manière arbitrairement grande. Nous utilisons ensuite la fonction de test *member* ainsi que *string-split* (notre implémentation qui utilise des fonctions de type fold) pour partitionner et modifier sa forme.

Lorsqu'il s'agit de l'ajout d'un mot à définition simple, la définition du noeud est simplement la liste de charactères après le signe '=' écrites par l'usager. Néanmoins, la définition peut être une concaténation de définitions de mots déjà existants dans l'arbre, ce qui demande un traitement supplémentaire. Plusieurs fonctions appelées l'une à la suite de l'autre permettent de trouver toutes les définitions des mots de la concaténation et de copier leur référence dans la définition du nouveau mot (ou du même mot s'il s'agit en même temps d'une redéfinition). De cette manière, nous n'avons pas à nous soucier de perdre une définition suite à une concaténation dans une mauvaise situation, puisque le récupérateur automatique de mémoire intégré à Scheme (garbage collector en anglais) sait qu'une référence existe dans un autre noeud de notre arbre si tel est le cas.

2.2 Représentation du dictionnaire et des définitions et opération sur ces structures

Le langage Scheme est un langage de type fonctionnel ; dans la partie du programme que nous avons fait (i.e. pas la partie fournie), nous utilisons un style fonctionnel pur car aucune des fonctions n'a d'effet de bord.
Nos noeuds sont représentés par une liste de quatre élément : l'enfant de gauche, la clé, la définition et l'enfant de droite. Les enfants sont soit nul, soit des noeuds. Notre dictionnaire peut être représenté par le noeud à la racine puisque celui-ci contient, à différente profondeur, tous les noeuds de l'arbre. Avec la même logique, chaque noeud représente le sous-arbre dont il est la racine. Il est à noter que ce dictionnaire est un représentation valide d'un arbre binaire. Quant aux définitions et clés, elles sont représentées par des listes de caractère. Les toutes petites fonctions suivantes nous permettent de mieux opérer sur les noeud et sur leurs attributs.

```
(define node-key cadr)
(define node-definition caddr)
(define node-lchild car)
(define node-rchild cadddr)
(define (node-create key definition lchild rchild) (list lchild
    key definition rchild))
(define (node-reconstruct x lchild rchild) (list lchild (
    node-key x) (node-definition x) rchild))
```

Pour le parcourir, on n'a qu'à aller dans le car ou le cadddr du noeud associé à la racine. Il est à noter qu'un parcours infixe de l'arbre correspond à un parcours en ordre alphabétique du dictionnaire.

Pour rajouter des noeuds, nous parcourons récursivement ces deux sous structures jusqu'à trouver le sous-arbre nul qu'il va remplacer pour devenir une feuille. Pour enlever des noeuds, nous reconstruisons l'arbre au fur et à mesure que nous recherchons le noeud. Une fois trouvé, nous retournons l'enfant du noeud lorsqu'il n'en a qu'un seul ce qui aura comme effet que l'enfant prendra la place du noeud supprimé. Dans le cas où il y a deux enfants, nous retournons à la place du noeud supprimé le résultat de l'insertion de l'enfant de droite dans l'enfant de gauche.

Pour ce qui est du splay, la fonction récursive *node-splay* prend en entrée la racine de l'arbre et la clé correspondante au noeud à mettre à la position de la racine. Cette fonction est appelée après chaque insertion et chaque recherche.

2.3 Affichage des réponses aux requêtes

La fonction non-modifiable go fournie par le professeur est responsable d'afficher les réponses de notre programme aux requêtes de l'usager. Pour ce faire, notre fonction principale traiter retourne une paire constituée du message à écrire à l'écran et du nouveau dictionnaire, possiblement changé suite à la requête. Par l'intermédaire de la fonction traiter-ligne, la paire est transmise à la fonction go qui lit ensuite tous les charactères du premier élément de la paire, comme on peut le constater dans le code de la fonction go :

```
          (let ((r (traiter-ligne ligne dict)))
            (for-each write-char (car r))
```

Par exemple, si l'usager demande à faire la recherche d'un mot et que celui-ci n'existe pas, le dictionnaire ne change pas et nous retournons la paire suivante à la fonction traiter-ligne :

```
                (cons (string->list "entree␣non-valide\n") dict)
                   )))))
```

2.4 d) Traitement des erreurs

Nous n'avons vu aucun type de traitement d'erreur en classe et n'en connaissions pas à la base pour le langage Scheme. Par conséquent, nous avons tout simplement décidé d'utiliser beaucoup de testage conditionnel pour différencier et traiter le plus de cas possibles. Certaines fonctions traitent les erreurs et d'autre retourne #f ou la liste vide, tout dépendant de la nature de leur opération. Puisque nous pouvons prévoir toutes les erreurs qui peuvent survenir selon les fonctions que nous utilisons, nous pouvons tester chaque opération que

nous faisons pour voir si une erreur est survenue. Il faut néanmoins être très pointilleux et très bien connaître tous les cas limites. Par exemple, la première chose que nous faisons dans notre fonction traiter est de vérifier si l'entrée de l'usager est vide, car dans ce cas aucun calcul n'est nécessaire :

```
(define traiter
  (lambda (expr dict)
   ;;;evaluer l'expression
   (if (null? expr) (cons (string->list "entree␣vide\n") dict) ;
       ;;l'utilisateur a taper enter et rien d'autre.
```

3 Comparaison des langages C et Scheme

En ce qui a trait à l'analyse syntaxique des requêtes, aucun des langages ne surpassent l'autre, les tests étant sensiblement les mêmes. Pour ce qui est du typage, bien que difficile à cerner au début, la représentation de tout en liste par Scheme s'avère utile et facilite les opérations sur les données. Néanmoins, le C avait l'avantage de nous permettre de faire pas mal n'importe quelle opération quand on le voulait, avec les effets de bord désiré. En effet, les langages qui intègre la programmation impérative nous permettent cela. Un langage de type fonctionnel, et même fonctionnel pur dans notre cas particulier, nous oblige a intégrer les opérations dans des fonctions et de faire la composition de fonction pour faire plusieurs opérations.
Dans la même lignée, le C nous a simplifié la vie en nous permettant de créer et de modifier des variables quand on le voulait, où on le voulait. Comment ajouter un mot au dictionnaire ? Simplement en changeant directement les pointeurs de l'enfant auquel il se rattache. En fait, peu importe l'action à faire dans le dictionnaire, le modifier ne consiste qu'à modifier directement la variable qui le contient. En Scheme, au contraire, il nous faut utiliser le retour de fonctions pour le reconstruire, ce qui n'est pas particulièrement plus compliqué mais inhabituel par rapport à notre expérience de codage. En plus, ce style de codage nous assure qu'aucun effet de bord ne sera produit. Cette particularité des langages fonctionnels est à double tranchant. D'une part, elle complique la vie au programmer, surtout celui habitué à coder de manière impérative depuis sa plus tendre enfance. De l'autre, elle simplifie la compréhension du comportement des fonctions du programme et améliore la réutilisation des fonctions.

Comme vous vous en doutez, l'utilisation de fonctions est primordial en programmation fonctionnelle. Le passage de fonction comme paramètre à d'autres fonctions s'avère un atout majeur que les langages comme C n'ont pas. Ceci permet notamment l'utilisation de foldr et de foldl, deux fonctions sensiblement pareilles qui appliquent récursivement des fonctions sur les éléments d'une liste. Ainsi, pour applatir une liste, en Scheme, nous utilisons la fonction suivante :

```
(define (construire-def lst)
        (foldr
                (lambda(x y)
                        (append x y)
                )
                '()
                lst
        )
```

En C, il aurait fallu faire une boucle, réfléchir aux bornes de la boucles, etc.

Sur le même ordre d'idée, le string-split des deux travaux pratiques représentent bien l'avantage du foldr. Dans le premier, nous avons du utiliser des tokens, dont l'utilisation demandait un doctorat en la matière. Dans le deuxième travail, nous avons tout simplement utiliser un foldr où la seule petite complication était de penser à la fonction à lui passer en paramètre, comme le montre le code suivant :

```
(define string-split
 (lambda (str chr)
  (foldr
    (lambda (x y)
            (if (equal? x chr)
                (cons '() y)
                (cons (cons x (car y)) (cdr y))))
    '(()) str)))
```

On peut voir que notre programme C a été écrit en environ 350 lignes contre environ 400 pour notre programme Scheme. En considérant qu'en C, nous n'avions pas implanté les concaténations et qu'en Scheme, nous avons ajouté beaucoup de test d'assertion (60 lignes) et nous avons implémenté le splay (90 lignes avec les commentaires), on conclut que Scheme est un langage plus concit et avec l'avantage indéniable d'avoir une gestion de mémoire automatisée (difficulté majeure dans le tp1).

Parlons maintenant un peu des fonctions itératives de notre programme. La fonction principale qui s'occupe du splay n'est malheureusement pas itératives. Elle s'appelle récursivement, mais ses appels récursifs ne sont pas en position terminal. En fait, la majorité, sinon l'entièreté de nos fonctions récursives ne sont pas sous formes itératives, parce que ce n'est pas naturel de programmer ainsi et que le programme fonctionne très bien sans. Toutefois, nous utilisons très souvent des foldr et des foldl dans nos fonctions (non-récursives), et ceux-ci sont sous formes itératives. Tout n'est pas noir !

En bout de compte, nul n'est meilleur que l'autre, mais chaque type de programmation a clairement des avantages et des inconvénients. Certains sont plus adaptés que d'autre pour des tâches particulière ; Scheme l'est définitivement plus pour l'implantation d'un arbre binaire. (Vincent préfère nettement Scheme et grogne des injures sur la gestion de mémoire en C)