Rapport du Travail Pratique 2

(TP2)

Par

Sulliman Aïad et François Poitras

Rapport présenté à

M. Marc Feeley

Dans le cadre du cours de

Concepts des langages de programmation (IFT2035)

Remis le 19 janvier 2017

Université de Montréal

Table des matières

1 Fonctionnement du programme

Tout comme dans le TP1, le programme est une calculatrice à précision infinie en notation postfixe. Le programme débute dans un mode d'attente, avec le signe « ?», pour différencier des instructions de l'interpréteur Gambit. L'utilisateur peut entrer une expression postfixe, demander la valeur d'une variable, ou encore affecter le résultat d'une expression à une variable. Notons qu'il est possible d'utiliser des variables dans les expressions et notons que les variables peuvent seulement être des lettres minuscules.

En cas de problème, soit avec la syntaxe ou avec l'utilisation de variables non-initialisées, l'utilisateur est informé de l'erreur qu'il a commise et le programme attend la prochaîne instruction. Si des variables étaient utilisées avec une mauvaise syntaxe, leur valeur ne sera pas initialisée ou modifiée.

2 Problèmes de programmation

2.1 Traitement d'une expression de longueur quelconque

Le traitement d'une expression commence par l'appel à la fonction *process* avec une liste vide comme premier argument (la pile) et un dictionnaire vide comme deuxième argument (qui contiendra les valeurs des variables). Notons aussi que l'autre argument de la fonction est l'expression elle-même, traitée par la fonction *split*. Cette dernière fonction a pour effet de transformer une chaîne de caractères en une liste. Pour ce faire, elle utilise une récursion en forme itérative. Un autre argument à la fin permet de distinguer un appel interne récursif de l'appel final, afin de formater le résultat en fonction.

La fonction *split* utilise un prédicat pour évaluer si le caractère courant est à utiliser comme séparation ou non. Dans le cas qui nous intéresse, le caractère de séparation est un espace, ce qui signifie que le prédicat est la fonction pré-définie *char-whitespace ?*. Une fonction lambda interne est appelée récursivement avec une liste vide, la liste à construire et une autre liste vide. Les deux listes vides représentent respectivement le résultat total et le mot en cours de traitement.

Si il ne reste plus rien à traiter et que le mot en cours est également vide, cela signifie que tout a été traité et on peut donc retourner la liste complète. Si la liste en cours n'est pas vide, on *cons* l'inverse de la liste en cours à l'inverse de la liste complète. Cela est nécéssaire car les caractères sont ajoutés au début de la liste complète. Pour avoir un résultat qui n'est pas inversé, on doit inverser la liste complète. Un raisonnement similaire explique l'inversion de la liste en cours.

D'un autre coté, s'il restait des éléments à traiter, on regarde si le caractère courant valide le prédicat. Si oui, on fait un appel récursif de la fonction lambda, en lui passant, dans l'ordre, la liste complète, la concaténation de l'inverse et de la liste en cours et le *cdr* de ce qui reste à traiter. Si le prédicat n'est pas vérifié, l'appel récursif est fait avec la liste complète, le *cdr* de ce qui reste et la concaténation du caractère courant à la liste en cours.

Tous ces appels récursifs font en sorte qu'une fois l'expression entièrement parcourue, la liste complète contient tous les mots de l'expression, selon le prédicat passé en argument. On peut alors passer au calcul de l'expression.

2.2 Calcul de l'expression

Le calcul se fait de manière récursive, Scheme oblige. La fonction *process* vérifie tout d'abord s'il reste des éléments à traiter. Si il n'y en a pas, elle vérifie le nombre d'éléments sur la pile. S'il est supérieur à 1, cela signifie qu'il y a une erreur syntaxique dans l'expression. S'il y a un seul élément, celui-ci est le résultat du calcul et il est affiché. Si aucun élément n'est présent, cela signifie qu'aucun calcul n'a été demandé.

S'il reste des éléments à traiter, on regarde la nature de l'élément courant. Si c'est un nombre, on l'enregistre sur la pile et on passe au traitement du prochain élément. S'il s'agit d'un opérateur, on s'assure qu'il y a suffisamment d'éléments sur la pile avant de dépiler deux fois et d'empiler le résultat du calcul.

2.3 Affectation et utilisation des variables

Toujours dans la fonction *process*, nous arrivons à traiter les variables. Si le mot courant est le signe "=" suivi d'une lettre, on affecte la valeur contenue sur la pile à cette variable et on appelle récursivement le prochain mot. Il est aussi possible qu'une variable soit dans le calcul. Si c'est le cas, on utilise *assoc* pour récuprerer sa valeur dans le dictionnaire. Si la variable demandée n'a pas de valeur, on affiche un message d'erreur. Le dictionnaire est toujours passé en paramètre parmi tous les appels récursifs, et dans l'appel final (qui se produit en fin d'une commande), on retourne le dictionnaire mis à jour si la commande réussit. Sinon, on retourne le dictionnaire de la dernière commande (ou du début, vide, si c'est la première commande).

2.4 Affichage des résultats et traitement des erreurs

Si tout s'est bien déroulé, on affiche le résultat du calcul ou la valeur de la variable demandée et le programme attend la prochaine expression. Si une variable a été affectée suite à un calul, on affiche aussi le résultat dudit calcul. Dans le cas où aucune des conditions plus haut n'a été vérifiée, cela signifie qu'une erreur s'est glissée dans l'expression. Un message informe l'usager de cette erreur et le programme se met en attente d'une nouvelle expression. Il en va de même pour l'appel à une variable non-initialisée.

3 Comparaison entre C et Scheme

L'utilisation d'un paradigme strictement fonctionnel a posé quelques difficultés au niveau du traitement des expressions, mais a été très bénéfique dans la gestion globale des ressources. Notre code Scheme est également beaucoup plus court. À titre de comparaison, notre code C comporte environ 1000 lignes

de code, contre 150 pour Scheme. Cette différence s'explique surtout par le caractère intrinsèquement récursif de Scheme et par l'absence de gestion de pointeurs.

Dans le language fonctionnel, il a été trivial d'implémenter des nombres de longueur arbitraire, car cette fonctionalité est déjà incluse dans Scheme, tandis qu'en C, il nous a fallu créer une structure pour gérer des nombres potentiellement très grands, avec tout les problèmes qui viennent avec. Ces problèmes incluent la gestion manuelle de la mémoire et l'utilisation de pointeurs. Par contre, étant donné la philosophie minimaliste de Scheme, il nous a fallu implémenter nous même des fonctions de base présentes dans les extensions autorisées dans le TP1. La fonction *split* est un bon exemple d'une telle fonction. En C, nous avions *strtok* pour nous aider. Dans un language impératif, implémenter cette fonction aurait été plus simple, car un traitement non-récursif est plus simple à implémenter. Sans doute qu'avec assez d'habitude, ce genre de problèmes seraient triviaux à résoudre, mais dans le cas de notre TP, nous n'avions pas encore cette aisance.

De l'autre coté, la difficulté en C a été de gérer manuellement des pointeurs et la mémoire sur des structures que nous avions nous-même créées. Il a fallu être extrêmement vigilants sur l'utilisation des *malloc* et des *free*. En particulier, nous avons prêté attention à libérer la mémoire au bon moment, que ce soit dans la gestion de la pile ou dans la gestion de très grands nombres.

Dans les deux languages, il nous a fallu faire une structure de pile. En Scheme, il s'agit d'une simple liste que nous utilisons récursivement, à l'aide des fonctions *car, crd, caar, cadr, etc.*. Il est relativement simple de dépiler en appelant la suite de la liste. Pour empiler, on peut utiliser *cons*. En C, la pile est une structure comportant une taille réelle, une taille maximale et des données. Pour empiler, il faut utiliser un calcul de pointeurs en créeant un élément au premier index disponible dans les données. Pour dépiler, le processus inverse est utilisé, mais au lieu d'allouer, on libère de la mémoire.

La dernière principale différence concerne la lecture de l'entrée. En C, on lit caractère par caractère et on alloue dynamiquement la mémoire au besoin. Ensuite, on peut commencer à traiter l'expression en utilisant *strtok*. Les résultats peuvent être affichés d'un coup avec des *printf*. En Scheme, chaque caractère est affiché à l'aide de la boucle *for-each*.