Université de Montréal

IFT2035-A2016

Rapport TP #2

Philippe CARON

Gabriel LEMYRE

Travail remis à l'intention de

Marc FEELAY

12 décembre 2016

1 Fonctionnement général du programme

Notre programme est une calculatrice à précision infinie à l'intérieur d'une console et programmée en scheme. Elle accepte des entrées de longueur arbitraire sous forme postfixe et permet l'enregistrement de valeurs dans plusieurs variables. Trois autres opérations sont aussi disponibles et peuvent être enchaînés. Premièrement l'addition nécessitant deux entrées préalables et présentant comme troisième valeur un symbole "+". La soustraction, de nature similaire mais demandant un symbole "-" au lieu du symbole plus. Enfin, la multiplication nécessitant elle aussi deux autres entrées, ainsi que le symbole "*" en troisième position. L'utilisateur doit espacer ses valeurs, et appuyer sur "Entrée" lorsqu'il a terminé la saisie de son opération, afin d'utiliser correctement ce logiciel. L'usager est invité à faire une entrée d'opération avec la chaîne "#". Les résultats sont affichés après chaque saisie d'expression de l'utilisateur.

Le programme utilise principalement un système de récursion afin de garder en mémoire le dictionnaire des variables, utilisant l'aspect fonctionnel plus que l'aspect impératif du langage. Le dictionnaire est une liste composée de liste de taille 2 utilisant un String représentant la variable et un String représentant la valeur.

2 Problèmes de programmation

2.1 Analyse syntaxique et traitement

Dans cette version de la calculatrice, l'analyse syntaxique et le traitement sont effectués en une seule étape. Le langage postfixe se prête très bien à ce fonctionnement étant donné que les valeurs analysées sont placées sur un stack. Ceci permet à l'analyseur syntaxique d'effectuer l'opération directement sur le stack plutôt que d'attendre que l'analyse soit terminée. De plus, le langage Scheme se prête énormément bien à l'implémentation d'une calculatrice à précision infinie puisqu'il supporte lui-même une précision infinie. Ainsi on peut tout simplement convertir les nombre lu en nombres Scheme et utiliser les fonctions natives d'addition, soustraction et multiplication (ce qui était impossible en C). L'implémentation des listes en Scheme rend également l'implémentation du stack très simple. Une expression typique est traitée comme suit par `parse` :

- La fonction lit des caractères correspondant à des chiffres qu'elle garde dans une variable
- Lorsqu'un espace est rencontré, elle converti la liste de chiffre en un nombre Scheme (en passant par un string)
- La fonction crée un autre nombre de la même manière
- La fonction lit un caractère correspondant à un opérateur
- La fonction modifie le stack en fonction de cet opérateur en utilisant les fonction native de Scheme
- Dans le cas ou une variable est assignée, une nouvelle entrée est créée ou modifiée dans le dictionnaire (sans effet de bord).

2.2 Calcul de l'expression

Le calcul d'un expression est fait au fur et à mesure de son analyse. Lorsqu'un opérateur est lu, le programme passe dans un case qui applique la fonction Scheme correspondante aux deux derniers

éléments du stack, puis remet dans le stack le résultat.

2.3 Affectation des variables

L'affectation des variables se fait en propageant un dictionnaire qui est modifié sans effet de bord lorsque le symbole «#
=» est lu. C'est pourquoi une opération erronée annulera un ajout ou une modification de variable effectuée dans une section précédente du code, même si il n'y avait pas de faute à ce moment.

2.4 Affichage des résultats et des erreurs

Les résultats et les erreurs sont affichées par une fonction fournie dans la donnée du TP qui requière une liste de caractère. La fonction `traiter` fourni donc une liste de caractère à celle-ci. Les nombres sont automatiquement convertis en listes de caractères tandis que les erreurs sont transmises sous forme de symboles. Quand le programme trouve une erreur il cesse immédiatement l'exécution et retourne le code d'erreur correspondant.

2.5 Traitement des erreurs

Si le programme détermine que l'exécution ne s'est pas produite comme il le faut, il retournera un symbole correspondant à l'erreur en cause, la fonction traiter déduira qu'il y a eu une erreur et retournera le dictionnaire original plutôt que le nouveau. Le programme pourra alors évaluer la valeur du symbole pour obtenir le code d'erreur.

3 Comparaison de l'expérience de développement

La calculatrice C comportes 956 lignes, tandis que la calculatrice Scheme n'en comporte même pas 100, presque 10% ! Quoiqu'il s'agisse d'une différence attendue, c'est une différence impressionante qui en dit long sur l'efficacité de la programmation fonctionnel dans ce cadre précis. C étant un langage impératif, il utilise des lignes d'assignation, chose que l'aspect fonctionnel de Scheme ne permet pas. De manière général, pour ce cas précis le langage Scheme est beaucoup plus approprié que C et permet une manipulation plus efficace et sécuritaire du code. Un des avantages indéniable de Scheme par rapport à C est la «garabage-collection» automatisée, ce qui permet d'abandonner des listes de temps en temps sans se casser la tête. L'aspect fonctionnel et le traitement des listes permet également de gérer un stack avec beaucoup de facilité.

Un aspect très bénéfique de la programmation fonctionnel est qu'il est très simple de traduire le pseudocode directement en prototype fonctionnel. Il est aussi très bénéfique lors de l'utilisation des opérations binaires et nous a réellement facilité la tâche pour leur implémentation. Par contre, l'assignation des variables dans le dictionnaire et la gestion des listes était plus difficile que dans un paradigme impératif. En effet, le désavantage principal de Scheme (son sous-ensemble fonctionnel du moins) par rapport à C est qu'il ne peut pas stocker de variable globale. Cela signifie qu'à chaque fois qu'une fonction veut utiliser une variable d'une autre fonction, celle-ci doit être passée en paramètre puis retournée d'une manière ou d'une autre. Ceci crée des fonction avec un nombre élevé de paramètre. Malgré tout, la syntaxe de Scheme rend le nombre élevé de paramètres très tolérable, et le jeu en vaut la chandelle.

Pour ce qui est de formes utilisées, la fonction `parse` utilise la forme itérative étant donné que ses appels récurents sont terminaux, de même que la fonction `assv-set`. Les fonction `variable->value`, `operate` et `traiter` sont toutes trois de simples conditions qui n'engendrent pas d'appel récursif.