Université de Montréal

Rapport TP 2

Philippe CARON
Gabriel LEMYRE

Travail remis à l'intention de:
Marc FEELAY

Décembre 2016

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.

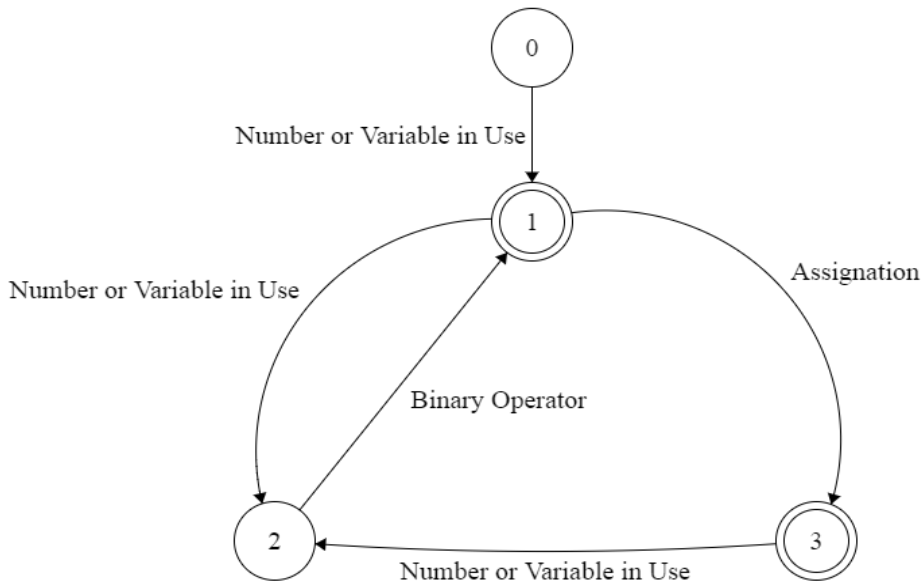
Le programme utilise principalement un système de récursion afin de garder en mémoire le dictionnaire des variables, utilisant l'aspect fonctionnel plus que l'aspect impératif du langage. Le dictionnaire est une liste composée de liste de taille 2 utilisant un *String* représentant la variable et un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

Résolution des problèmes de programmation

Analyse syntaxique et traitement d'une expression de longueur quelconque

Notre programme transforme l'expression entrée par l'utilisateur en liste de *String*. Tout d'abord, le programme lis les caractères fournis par l'usager et les place dans une liste. Ceci est effectué dans la méthode *repl* fourni par le professeur. Nous utilisons ensuite cette liste de caractère dans notre méthode *string-split*, appelant *createStringList*, afin de transformer chaque entrée séparé par un caractère *#/space* en élément de la liste de string représentant l'expression. Ainsi, une entrée telle "147 145 +" sera représentée comme *(list "147" "145" "+")*. Scheme n'utilisant pas de limite de taille pour les entiers, un traitement par caractère n'est pas nécessaire.

Afin de vérifier si l'opération est d'une syntaxe acceptable, notre méthode *formatAutomata* suit le processus de l'automate suivant (l'état initial est 0) :



Chaque élément de la liste de l'expression permettant une progression au travers cet automate. L'entrée est acceptée si l'automate se termine dans les états 1 ou 3. Par exemple, une entrée "12 37 * =a 74 + =b a +" nous fera parcourir les états dans cet ordre : 1, 2, 1, 3, 2, 1, 3, 2, 1. S'arrêtant ainsi sur l'état acceptant 1. L'expression de l'utilisateur est donc valide et traitable.

Calcul de l'expression

Après avoir validé l'expression à l'aide de notre automate, nous appelons la méthode *nextOperation* qui détermine quelle est la prochaine opération à effectuer. Elle vérifie en premier le nombre d'éléments dans la liste. Si la liste ne comporte qu'un seul élément, il s'agit de la réponse et est donc retourné. Sinon, elle vérifie s'il s'agit d'une assignation de variable. Si tel est le cas, nous passons directement à l'assignation. Sinon, nous vérifions quel est l'opérateur binaire à la troisième position de la liste. S'il s'agit d'une addition, nous appelons la méthode *add* afin d'additionner les deux premiers éléments de la liste et les remplacer ces trois éléments par la somme des deux premières valeurs. Dans les autres cas, les méthodes *sub* ou *mul* sont appelées et effectuent le même genre d'opérations, excepté pour une soustraction et une multiplication au lieu d'une addition.

Affectation de variables

Les variables sont assignées deux fois. La première fois à l'intérieure de l'expression afin de pouvoir les utiliser dans ce cas, et la seconde fois afin qu'elles entrent dans le dictionnaire afin de donner la possibilité à l'utilisateur de les réutiliser pour ses prochaines expressions. La méthode assign retourne une liste composée de l'expression contenant la valeur assignée moins la chaîne utilisée pour l'assignation et le dictionnaire. À ce qui a trait

au dictionnaire, si l'assignation s'effectue pour une variable ayant déjà une valeur, la valeur initiale est retirée de la liste et la nouvelle valeur est ensuite ajoutée. S'il s'agit toutefois d'une nouvelle variable, elle est simplement ajoutée au dictionnaire.

Affichage des résultats et des erreurs

Nous n'avons rien eu à programmer pour faire l'affichage des résultat. Elle est entièrement gérée par la méthode repl qui nous a été fournie. Toutefois, nous fournissons à cette méthode une liste de string qu'elle affichera à l'écran caractère par caractère, composée d'éléments d'affichage tels le résultat, l'expression lue et le nombre de caractères.

Traitement des erreurs

Afin d'éviter une assignation fautive de variable au dictionnaire et pour faciliter le traitement des opérations, les erreurs sont détectées grâce à l'automate *formatAutomata* de la page 2. Cet automate retournera une valeur booléenne *#f* en cas d'erreur ou lorsque l'état final ne sera pas acceptant. Les erreurs affichées sont donc assez génériques.

Comparaison de l'expérience de développement

La calculatrice *C* comportes 956 lignes, tandis que la calculatrice *Scheme* en comporte 643. Il s'agit d'une différence attendue, *C* étant un langage impératif, il utilise des lignes d'assignation, chose que l'aspect fonctionnel de *Scheme* ne permet pas.

Les éléments les plus faciles à implémenter en Scheme sont clairement les opérations binaires. Il ne nous a fallu que très peu de temps afin de comprendre comment utiliser ses opérations à bon escient et ainsi faire avancer notre projet. Toutefois, les aspects qui ont été les plus contraignants à exprimer en scheme sont le transfert du dictionnaire des variables et la vérification de la chaîne. Le transfert du dictionnaire nous a causé quelques maux de têtes puisqu'il fallait trouver un moyen de permettre à l'utilisateur de pouvoir l'utiliser pour ses futures expressions, ce en utilisant du code déjà existant. Aussi, la vérification de la chaîne fût très ardue à compléter. Nous avons dû réécrire de bout de code au moins trois fois avant d'opter pour l'utilisation d'un automate. Cette décision nous a énormément aidé et nous a permis de terminer cette section en très peu de temps.

Un aspect très bénéfique de la programmation fonctionne 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. Il nous fallait, en effet constamment modifier la liste avant de faire une récursion ou lors d'un autre appel de fonction. Nous souffrions aussi d'un problème similaire pour le dictionnaire.

L’automate de vérification utilise une itération afin de modifier les états. La méthode *nextOperation* est semi-récursive puisqu’elle est appelée par les méthodes *add*, *sub, mul* et *assign* qu’elle appelle elle-même. La fonction *createStringList* est aussi récursive et est appelée par string-split.