Université de Montréal

Devoir 2

Par

Catherine Laprise

Farzin Faridfar

Baccalauréat en informatique

Faculté des Arts et des Sciences

Travail présenté à Marc Feeley

Dans le cadre du cours IFT2035

Concepts des langages de programmation

12 décembre 2016

1 - Fonctionnement général du programme

La fonction principale du programme est *lire-expr-aux*, qui est elle-même une fonction auxiliaire de *lire-expr* (ce qui permet d'utiliser la forme itérative de la fonction). *Lire-expr-aux* se charge d'analyser l'expression entrée un char à la fois, d'empiler les opérandes lus sur la pile et d'appeler les fonctions nécessaires aux diverses opérations prévues par la calculatrice (opérations arithmétiques, assignation de variable, gestion des erreurs, etc.).

La fonction *operation* effectue toutes les opérations arithmétiques. Scheme nous permet de passer directement l'opérateur désiré en paramètre à cette fonction et ainsi l'utiliser pour tous les cas de figure.

Affectation est la fonction principale en ce qui concerne l'affection de valeur aux variables via le dictionnaire. Si la variable entrée n'est pas déjà dans le dictionnaire (incluant le cas où celui-ci est vide), la nouvelle paire clé-valeur y est ajoutée. Sinon, on fera appel à *creer-dict* et *créer-dict-aux* afin de remplacer la valeur d'une clé existante.

Finalement, *exception* sert à retourner le message d'erreur approprié en fonction de l'erreur commise. Cela nous permet d'afficher des messages détaillés et offre plus de robustesse au programme.

2 - Résolution des problèmes

- Analyse syntaxique et traitement des expressions de longueur quelconque

L'analyse syntaxique se fait dans la fonction *lire-expr-aux*, qui lit chaque char de l'expression entrée. Cette fonction reçoit cinq arguments :

- *expr* - la liste des caractères contenant l'expression lue
- *init_dict* - la liste d'association des variables telle qu'elle est au début de la lecture de la ligne
- *updated_dict* – la liste d'association des variables mise à jour en cours de lecture
- *tmp* - une chaîne de caractère contenant les chiffres lus en format string
- *pile* - la liste contenant les opérandes lus sur la ligne

La fonction effectue des traitements différents selon le type de caractère lu.

Pour la lecture des nombres (type *char-numeric*), les caractères lus sont emmagasinés dans le string *tmp*. Ceci est réalisé en appelant récursivement la fonction sur *cdr expr* et en mettant à jour l'argument *tmp* avec le nouveau caractère lu. La fin du nombre est détecté quand le caractère '#\space' est lu. On rappelle donc récursivement la fonction avec la chaîne vide comme argument *tmp* et on ajoute le contenu actuel de *tmp (*converti en nombre avec la fonction *string->number*) à l'argument *pile.* Ces étapes continuent récursivement jusqu'à ce qu'une opération, une lettre ou le symbole d'égalité soit entré.

Dès qu'un opérateur (#\+, #\-, #*) est lu, on appelle la fonction *operation* qui reçoit en paramètre l'opération lue et la pile d'opérandes.

Si un caractère alphabétique est lu (type *char-alphabetic*), on peut chercher cette variable dans la dictionnaire *updated_dict* en utilisant la méthode *assoc*. La valeur de la variable est ajoutée dans la pile si *assoc* retourne un résultat valide. Sinon, une erreur est lancée.

En lisant le caractère #\= la fonction *affectation* est appelée. Celle-ci est responsable de gérer le dictionnaire, soit en ajoutant une nouvelle paire clé-valeur, soit en modifiant la valeur d'une variable déjà existante. Le fonctionnement de cette fonction est précisé dessous dans la section « Affectation aux variables ».

Nous avons aussi ajouté le support de la commande « ,q » pour que l'usager puisse quitter facilement l'environnement de la calculatrice.

Toute autre expression est invalide et engendre une erreur.

- Calcul de l'expression

Comme Scheme permet les fonctions d'ordre supérieur, on peut directement passer l'opérateur désiré en paramètre, et toutes les opérations peuvent se faire à partir de la même fonction. Dans la fonction *operation,* on appelle donc l'opérateur *op* avec les deux premières valeurs de la pile comme arguments. Les deux opérandes dans la pile sont remplacés par le résultat et la nouvelle pile est retournée à la fonction lire-*expr-aux*.

- Affectation aux variables

En ce qui concerne le traitement des variables, les paires clé-valeur sont stockés dans un dictionnaire *dict* qui est une liste de listes impropres. Afin d'éviter tout changement permanent au dictionnaire en cas d'erreur de lecture, on passe en paramètre deux dictionnaires qui sont initialement identiques : *init_dict,* le dictionnaire initial, et *updated_dict,* le dictionnaire qu'on mettra à jour en cours de lecture. Si la ligne est lue en entier sans erreur, *updated_dict* deviendra le dictionnaire initial pour la prochaine itération de *lire-expr-aux.* Sinon, on renverra à nouveau *init_dict*, annulant ainsi tout changement effectué dans la ligne erronée.

Dans le cas de l'appel d'une variable en tant qu'opérande, on parcoure le dictionnaire pour trouver la paire correspondante avec la fonction *assoc*. Si le dictionnaire est vide ou si la variable n'y existe pas, une erreur est retournée à l'usager. Si la variable est trouvée dans le dictionnaire, sa valeur est empilée dans la liste *pile* pour qu'on puisse effectuer une opération.

Dans le cas où on souhaite affecter une valeur à une variable (avec l'opérateur =), on doit soit insérer une nouvelle paire dans le dictionnaire, ou remplacer une paire existante. La fonction *affectation* prend l'expression *expr*, le dictionnaire *dict* et la pile des opérandes *pile* comme arguments. Si le dictionnaire est vide ou s'il ne contient pas la variable cherchée, la paire (variable . valeur) est ajoutée directement dans le dictionnaire. Si la paire est déjà dans le dictionnaire, il faut mettre à jour sa valeur. Comme il n'est pas possible d'accéder directement à une paire pour en modifier la valeur, on doit créer un nouveau dictionnaire contenant la nouvelle paire et le reste de l'ancien dictionnaire, à l'exception de la paire modifiée. Ces opérations sont réalisées dans la fonction *créer-dict-aux* qui parcoure le dictionnaire et copie chaque paire dans un nouveau dictionnaire, en ignorant la paire à modifier et en ajoutant la nouvelle paire à la fin du dictionnaire.

- Affichage des résultats et erreurs

Le résultat (stocké dans la variable *lecture* dans la fonction *traiter*) est retourné comme une liste qui contient la valeur numérique de résultat d'expression dans le *car* ainsi que le dictionnaire à jour dans le *cdr*. Le programme peut ainsi conserver en mémoire le dictionnaire à chaque appel de fonction puisqu'il est passé en argument récursivement à chaque appel à *repl.*

Pour l'affichage des erreurs, on fait appel à la fonction *exception*, auquel on donne en paramètre le type d'erreur. Les messages d'erreurs sont des chaînes de caractères qu'on convertit en liste avec la méthode *string→list*. On peut donc passer cette liste comme résultat avec le dictionnaire initial pour que la fonction *traiter* puisse les afficher sans interrompre la boucle *repl*.

- Traitement des erreurs

Les erreurs sont traitées dans la fonction *exception* qui prend un message et l'expression en cours de lecture comme arguments. Le message spécifie à la fonction d'où vient l'erreur pour retourner un message précis à l'usager afin qu'il corrige sa requête (une fonctionnalité comme *switch* en C).

Dans le cas d'une affectation ou appel de variable erroné, l'expression *expr* est utilisée pour préciser la variable qui n'est pas entrée correctement par l'usager.

Comme mentionné précédemment, le message d'erreur et le dictionnaire initial sont retournés comme résultat à la fonction *traiter*. Ainsi en cas d'exception, toutes les nouvelles affectations ajoutées lors de la lecture de la ligne en cours seront ignorées. Ainsi on retourne à l'état original du dictionnaire avant la lecture de la ligne.

Les erreurs considérées sont les suivantes :

- Il manque des opérateurs ou des opérandes pour compléter l'expression;
- Une variable est appelée alors qu'elle n'existe pas dans le dictionnaire, ou que celui-ci est vide;
- Un nombre est entrée avec le symbole '=' sans aucun nom de variable;
- Le symbole '=' avec un nom de variable sont entrées sans avoir donné une valeur d'abord;
- Un espace de trop est entré après le symbole '=';
- L'expression entrée est autrement invalide.

3 - Comparaison avec le TP1

Combien de lignes de code ont vos programmes C et Scheme?

Le code du programme réalisé est beaucoup plus compact en Scheme qu'en C.

- En C : 873 lignes de code, une trentaine de fonctions
- En Scheme : 169 lignes de code, 11 fonctions

Sans tenir compte de votre niveau de connaissance des langages C et Scheme, quels sont les traitements qui ont été plus faciles et plus difficiles à exprimer en Scheme?

Un avantage important en Scheme est la gestion automatique de la mémoire. Nous avons épargné beaucoup de code puisque nous n'avions pas à nous soucier d'allouer ou libérer les blocs mémoires.

De plus, au niveau de la pile, il était facile d'effectuer les opérations car les listes sont déjà implémentées comme des listes chaînées. Nous n'avons donc pas eu à définir de structures particulières ou d'utiliser de pointeurs puisqu'on peut facilement accéder aux éléments via le *car, cdr* et autres variations. Nous avons pu également utiliser des fonctions prédéfinies comme *cons* et *append* pour gérer les listes. La conversion des nombre (qui peuvent prendre la forme de list, de string ou de nombre) est également facile à Scheme grâce à des fonctions comme *string->list* ou *string->number* qui sont prédéfinies dans le langage (alors qu'en C nous devions convertir les *char* en *int* à chaque étape et effectuer beaucoup de traitement pour obtenir le bon format).

Finalement, la gestion des grands nombres étant possible en Scheme, cela a simplifié grandement notre travail puisqu'on pouvait obtenir une très grande précision sans avoir à transformer les entiers en listes. Cela nous a permis d'utiliser les opérateurs directement sans avoir à coder les algorithmes de calcul comme nous l'avions fait en C.

Par contre il était moins évident de faire la gestion des erreurs en Scheme. En C, il était assez facile d'interrompre la boucle à tout moment en affichant un message d'erreur, alors qu'en Scheme nous avons dû créer une fonction qui retourne le message d'erreur comme résultat.

Le débogage était également parfois difficile car les erreurs de syntaxe sont parfois difficiles à détecter. Par exemple, nous avons éprouvé des difficultés avec la mise à jour du dictionnaire qui étaient causées simplement par une parenthèse qui était au mauvais endroit. Il nous a été difficile de détecter l'erreur puisque le programme fonctionnait quand même mais retournait des résultats erronés. Toutefois, ce problème est certainement dû en partie à notre manque de connaissance du langage.

Indépendamment des particularités de Scheme, pour quelles parties du programme l'utilisation du style de programmation fonctionnel a-t-il été bénéfique et pour quelles parties détrimental?

L'utilisation des fonctions d'ordre supérieur a été très pratique pour le traitement des opérations, puisqu'on a pu passer les opérateurs directement en paramètre et ainsi généraliser la fonction de calcul.

L'impossibilité de définir et modifier des variables globales pour tout le programme a toutefois compliqué les opérations, particulièrement au niveau de la gestion du dictionnaire et de la pile, puisqu'il fallait toujours pouvoir repasser en paramètre les données actuelles afin de s'assurer de ne perdre aucune information en cours d'exécution. De plus, comme nous devions utiliser la récursion, nous avons dû créer des fonctions supplémentaires afin de transformer nos fonctions en forme itérative pour éviter d'encombrer la pile durant la lecture de grands nombres.

Pour quelles parties avez vous utilisé des récursions en forme itérative? Pour quelles parties avez vous utilisé des continuations?

Nous avons utilisé la récursion en forme itérative pour la lecture des expression (*lire-expr et lire-expr-aux*) ainsi que pour la mise à jour du dictionnaire (*creer-dict et creer-dict-aux).*

Toutefois nous n'avons pas utilisé la continuation dans ce travail.