IFT2035 – Concepts des langages de programmation

Professeur : Marc Feeley

Travail Pratique #2

9 décembre 2016

Nom : Paul Chaffanet
Matricule : 1009543

Nom : Samuel Guigui
Matricule : 20035458

1) Fonctionnement général du programme

Le programme se lance après un appel à la fonction `main` qui ne prend aucun paramètre et qui fait appel à la fonction `repl` en lui passant en paramètre le dictionnaire stockant les variables, initialement vide. Cette fonction s'occupe respectivement de récupérer l'expression entrée par l'utilisateur, l'envoyer à `traiter` et afficher le retour de `traiter`. Elle ne s'arrête que si l'utilisateur entre en input *ctrl+c.*

Expliquons maintenant dans les grandes lignes la partie traitement de l'expression, assurée par la fonction `traiter`. Elle reçoit en paramètre le dictionnaire et l'expression sous forme de liste de caractère. Le traitement s'opère en deux étapes. Une analyse syntaxique des mots de l'expression puis l'évaluation.

L'analyse syntaxique des mots est assurée par la fonction `slice-expr`, qui s'occupe de découper en mot l'expression et de les valider. Pour valider les mots nous avons défini un langage `L` et ses règles de grammaire (`nombre`, `opbin`, etc…). Le langage vérifie uniquement la syntaxe d'un mot ; la syntaxe de l'expression dans son ensemble est vérifiée lors de la partie calcule. Au fur et à mesure que les mots sont validés, on les place dans une liste appelée `stack` qui sera envoyée à la fonction `evaluate-stack` qui s'occupe de la partie calcule.

`evaluate-stack` calcule l'expression postfix avec la méthode de la pile. Elle gère les affectations et s'occupe de détecter les erreurs suivantes :

- Expressions en forme non postfixe

- Variables inexistantes

- Affectations sans valeur.

Si la fonction `evaluate-stack` rencontre l'une de ces erreurs lors du calcul elle s'arrête et renvoie un message d'erreur approprié. Sinon elle retourne le résultat. `repl` s'occupera de l'affichage de l'erreur ou du résultat dans la console.

2) Résolution des problèmes de programmation

a) Analyse syntaxique de l'expression de longueur quelconque

L'analyse syntaxique de l'expression s'effectue en deux étapes :

Etape 1 : Découpage en mot valide de l'expression

La fonction `slice-expr` prend en charge cette opération. L'expression est découpée en mot avec comme délimiteur le caractère *'#\space*'. Chaque mot extrait de l'expression est inséré dans la pile si il est valide. Un mot est valide s'il appartient au langage des expressions postfixe que nous avons défini comme suit :

$$L_{postfix} = \{ [a-z], [0-9][0-9]*, `='[a-z], +, -, * \}$$

- Si un mot ne fait pas partie du langage la fonction `L` retourne **`#f`** et le découpage s'arrête.
- Si tous les mots sont validés, la fonction de découpage retourne une pile sous forme de liste comprenant tous les mots de l'expression. Cette pile sera ensuite utilisée pour calculer l'expression postfixe.

Par ailleurs cette fonction est récursive, c'est-à-dire qu'après la lecture et analyse d'un mot, elle se rappelle elle-même met en avançant d'un mot, et ce jusqu'à lecture de `null.` Ainsi l'expression peut être arbitrairement longue.

Note : les espaces sont ignorés c'est-à-dire que l'utilisateur peut entrer autant d'espace qu'il souhaite entre chaque mot. Par exemple ' 2 5 + =a ' sera reconnu comme expression valide.

Etape 2 : Validation de la forme de l'expression

Une fois l'expression découpée en mot valide, il faut vérifier que l'expression est calculable c'est-à-dire qu'elle est bien de la forme :

$$< expr > ::= < expr >< expr >< opbin > \{< affectation >\}* \\ | < value > \{< affectation >\}*$$

Pour y parvenir nous utilisons la pile avec les règles suivantes pour l'évaluer :
- Si on lit une variable ou un nombre on la/le push dans la pile

- Si on lit un operateur binaire, on vérifie que la pile contienne au moins deux éléments.
  - Si < 2 éléments → retourne **`#f`**, l'expression n'est pas sous la bonne forme, un opérateur binaire doit nécessairement avoir deux arguments
  - Sinon → on pop deux nombres de la pile et effectuons le calcul indiqué par l'opérateur binaire lu. Le résultat est ensuite push dans la pile.

- Si on lit une affectation on vérifie que la pile contienne au moins un élément.
  - Si < 1 élément → retourne **`#f`**, il n'y a aucune valeur à affecter, l'expression est de la mauvaise forme.
  - Sinon on affecte la valeur du dessus de la pile à la variable spécifiée dans le mot.

- Si tous les mots ont été lu, on vérifie que la pile contienne exactement un élément.
  - Si ≠ 1 élément → retourne **`#f`**, l'expression n'est pas sous la bonne forme.
  - Sinon → L'expression est validée et l'élément restant dans la pile est le résultat

b) Calcul de l'expression

Le calcul de l'expression se réalise en même temps que la validation de la forme de l'expression décrit ci-dessus. Tant qu'aucune des fonctions servant à calculer

l'expression ne retourne **#f**, le calcul continu. Nous avons déjà expliqué le fonctionnement de la pile précédemment. Intéressons-nous plutôt ici à la partie application numérique.

En supposant que l'expression est valide, quand on rencontre un opérateur binaire, il faut effectuer une application numérique. Pour cela, il faut 'pop' deux nombres de la pile et effectuer le calcul spécifier par l'opérateur binaire lu. Or, dans la pile, les nombres sont sous forme de liste et les variables ainsi que les opérateurs binaires sont des caractères. Par conséquent, pour effectuer un calcul il faut effectuer les transformations suivantes :

- Convertir les listes en nombres avec les fonctions prédéfinies `list->string` et `string->number.` De plus du fait qu'en Scheme les listes se comportent en fait comme une pile, quand on lit et découpe l'expression, les chiffres d'un nombre seront écrits à l'envers ('12' -> '21'). Ainsi nous utilisons la fonction prédéfinie 'reverse' pour corriger cet effet.
- Convertir les variables en allant chercher leur valeur dans le `dict`
- Convertit les opérateurs binaires en procédure. Pour cela, nous avons défini la fonction `opbin->proc` i.e. *'#\+'* → $< procédure + >$.

Une fois les transformations effectuées, on calcule le résultat puis on le 'push' dans la pile sans oublier de le transformer en liste préalablement.

c) Affectation des variables

Lors de l'évaluation de la pile, lorsque le mot lu est une affectation, on fait appel à la fonction `affectation`. Elle prend en paramètre le dictionnaire et l'association `(var . value)` ou `value` est la valeur du dessus de la pile et `var` est le caractère.

Initialement, elle vérifie si la variable à affecter existe déjà dans le dictionnaire.

- Si la variable existe, on la supprime du dictionnaire puis on l'ajoute à nouveau avec sa nouvelle valeur. Pour supprimer du dictionnaire l'ancienne version de la variable on fait une concaténation. Soit x la variable à supprimer : $concat\ \ [début, x[\ et\ ]x, fin]$
Nous avons défini la fonction `del-assoc` pour assurer cette opération.

- Sinon, on insert directement dans le dictionnaire la variable et sa valeur.

Finalement la fonction retourne le nouveau dictionnaire à la fonction qui calcule l'expression.

d) Affichage des résultats et erreurs

Pour afficher les erreurs il faut faire remonter jusqu'à la fonction de traitement principale le message d'erreur émis au moment de la détection. Ainsi les fonctions

qui détectent les erreurs renvoie soit le résultat de l'opération si aucune erreur n'est détectée soit une string contenant le message d'erreur. Par conséquent il faut ajouter une opération de contrôle dans la fonction de traitement principale qui vérifie si les retours d'appel de fonctions sont des strings ou non. Si c'est une string, elle arrête le traitement et la renvoie à `repl`.

L'affichage des résultats se fait uniquement si aucune erreur n'a été détecté. La fonction qui calcule l'expression avec la pile, renvoie une fois tous les mots lus, la pile qui contient forcement qu'une valeur étant le résultat de l'expression. On rappelle que dans la pile les nombres sont sous forme de liste, ainsi aucune conversion ne sera nécessaire car la fonction `repl` qui s'occupe de l'affichage console récupère l'information à afficher sous forme de liste.

e) Traitement des erreurs

Pour ne jamais 'planter', notre programme doit pouvoir être capable de gérer tous les types d'erreurs que peut engendrer l'entrée d'une expression erronée de l'utilisateur. Voici la liste des erreurs pris en charge par le programme et le détail de leur traitement :

- Erreurs de syntaxe :

Il y plusieurs types d'erreur de syntaxe. Le premier est la mal formation d'un mot. Par exemple : *'1f5', '++', 'ab', '!', ' ?' etc…*
Le langage `L` reçoit en paramètre l'expression, lit les caractères un par un et vérifie à chaque fois si l'ajout du caractère lu peut former un mot dans le langage. Si l'ajout d'un caractère fait sortir le mot du langage, `L` retourne `#f` à `slice-expr` qui arrêtera l'évaluation de l'expression.

Le deuxième type d'erreur de syntaxe est la mal formation de l'expression.
Par exemple : *'5 + 5', '4 4 4 +', '=a 5' etc…*
Une mal formation de l'expression peut mener à trois scenarios différents :

- **Un opérateur a moins que deux opérandes**. Ce scenario est traité par la fonction `operation` qui est appelé à chaque lecture d'un opérateur dans la pile qu'elle reçoit en paramètre. Si son nombre d'éléments est inférieur à deux, le calcul est impossible et elle retourne un message d'erreur qui arrête l'évaluation de l'expression.

- **Il reste plus d'une valeur après que tous les opérateurs soient lus**. Ce scenario est traité par la fonction `evaluate-stack`. Elle reçoit en paramètre une liste contenant les mots de l'expression. Si elle lit tous les mots sans qu'une erreur soit retournée, elle vérifie que la pile contienne qu'une seule valeur étant le résultat de l'expression. Retourne un message d'erreur si faux.

- **Une affectation n'a pas de valeur à affecter**. Aussi traité par la fonction `evaluate-stack` ; à chaque fois qu'elle lit un mot correspondant à une affectation (ex. *'=a*') elle vérifie si la pile n'est pas vide. Si faux, il n'y a aucune valeur à affecter donc arrêt de l'évaluation et retour d'un message d'erreur.

- Variable inexistante

Lorsque l'utilisateur entre des variables dans l'expression il faut s'assurer qu'elles existent. La fonction `evaluate-stack` traite cette erreur en vérifiant à chaque lecture d'un mot correspondant à une variable, son existe dans le dictionnaire à l'aide de la fonction (`assoc key dict`). Si `assoc` retourne **`#f`**, l'évaluation s'arrête et on retourne un message d'erreur.

3) Comparaison de l'expérience de développement

Ces deux expériences de développement ont été enrichissante bien qu'assez différente. Scheme a été pour nous une introduction aux langages fonctionnel, ce qui a rendu l'expérience un peu déroutante au début. En effet les langages fonctionnels ont une logique de programmation vraiment différente des langages impératif et OO dont nous sommes maintenant habitués et leur syntaxe est plus éloignée des langages naturels rendant leur utilisation moins instinctive au début.

Cependant, après adaptation il s'est avéré que pour cet exercice, le développement a été plus agréable qu'en C notamment grâce à l'élégance qu'offre Scheme, la prise en charge par les librairies de bases des big numbers et la gestion de mémoire automatique. Aucune difficulté liée à Scheme n'a été aussi lourde à résoudre que ces deux dernières expliquant en partie la différence de taille notable du code entre les deux langages. Notre code C a environ 1200 lignes alors que notre code Scheme n'en comporte que 200.

Traitement plus facile en Scheme :

- Le traitement de la mémoire est automatique
- Les librairies de base gèrent les big number donc le stockage et le calcul des nombres ne demande aucun algorithme particulier. Nous n'avons pas besoin comme en C de représenter un nombre par des structures chainées et d'implanter des algorithmes de 'calcul a la main' pour effectuer les opérations +, -, *.
- Recherche d'existence d'une variable facilitée par la liste associative de plus haut niveau que le système de stockage des variables utilisé en C.

Traitement plus difficile en Scheme :

- Gestion des opérations sur les listes de manière récursive
- Pas de déclaration directe des variables.
- Pas de mutation des données (`set!` etc… interdis). Pose problème lors de la mise à jour d'une variable dans le `dict` par exemple.

Par ailleurs ce travail pratique nous a permis de comprendre les avantages qu'offre le style de programmation fonctionnel. En effet la principale caractéristique de ce style est l'absence d'effet de bord. Ne pas pouvoir muter les données nous a paru au premier abord comme un handicape mais s'est finalement avéré être un sérieux avantage de robustesse par rapport au style impératif. Qui plus est, le

langage C est d'assez bas niveau ajoutant une difficulté supplémentaire a la gestion de la robustesse. Cette caractéristique implique un second avantage qui vient consolider l'argument de la robustesse. Vu qu'il n'y a pas de mutation, il faut découper le programme en petite fonctions qui sont donc très facile à tester de manière isolée. On peut toutes les tester individuellement en leur passant des paramètres manuellement. La recherche de la cause d'un bug est ainsi en pratique plus facile à localiser. De plus il devient plus facile qu'en style impératif de tester tous les états que peut rencontrer le programme augmentant ainsi la robustesse.

En revanche ce TP ne nous a pas permis de mettre à la lumière un réel désavantage du style de programmation fonctionnel par rapport à l'impératif. En tout point le style fonctionnel rempli le même travail que le style impératif mais de manière plus élégante et plus robuste.

Nous avons mis toutes nos fonctions récursives sous forme itérative. Pour un cas les continuations ont été nécessaire. La fonction `del-assoc` qui s'occupe de supprimer du dictionnaire une variable. Comme nous l'avons vu les données sont immutables donc on ne peut pas juste retirer du dictionnaire la variable, il faut reconstruire un nouveau dictionnaire sans la variable à supprimer. Au début la fonction était récursive mais retournait :

```
(cons (car dict) (del-assoc (cdr dict)))
```

Ce n'est pas un appel terminal car `cons` attend le retour de l'évaluation de `(del-assoc (cdr dict)` ce qui multiplie les blocs. Par conséquente la fonction n'était pas en forme itérative. Nous avons donc ajouté une continuation pour contourner le problème. Le retour est maintenant :

```
(del-assoc (cdr dict)
           (lambda (r) (cont (cons (car dict) r))))
```