IFT2035 - Concepts des langages de programmation

Professeur: Marc Feeley

Travail Pratique #1

14 octobre 2016

Nom: Paul Chaffanet
Matricule: 1009543

Nom: Samuel Guigui
Matricule: 20035458

1. Fonctionnement général du programme

Le programme est une calculatrice à précision illimitée. Le programme fonctionne de la façon suivante:

1. On scanne chaque caractère l'un après l'autre. Dès que les caractères analysés forment un mot, on analyse ce mot afin de savoir si c'est un littéral (un nombre entier ou une variable), un opérateur binaire (+, −, ∗) ou un opérateur unaire (?, =a).

2. Si ce mot n'est pas valide, on le rejette immédiatement.

3. Si ce mot est valide et est un nombre entier ou une variable, on le met dans la pile

4. Si ce mot est un opérateur binaire (+, −, ∗), alors on retire deux nombres de la pile et on évalue l'opération. Puis on met le résultat dans la pile.

5. Si ce mot est une opération unaire (?, =a), alors on retire un nombre de la pile, on effectue l'opération appropriée (obtenir le compteur de référence ou effectuer une affection). Puis on met également le résultat de cette opération dans la pile.

6. Chaque nombre n'apparaît qu'une seule fois en mémoire. On vérifie au moment de l'apparition d'un nouvel entier que celui-ci n'existe pas déjà en mémoire ou dans la pile. S'il existe, alors il suffit de pousser sur la pile le pointeur du nombre, et d'incrémenter son compteur de référence.

7. On répète également cette manière de faire lorsque qu'apparaît les résultats des opérations binaires et unaires, en vérifiant à chaque fois s'il n'existe pas une copie existante de ce nombre en mémoire ou dans la pile. Si oui, alors il suffit d'incrémenter son compteur de référence en le poussant dans la pile.

8. Si on pop et que la pile est vide, cela veut dire qu'il nous manque une opérande au moment de faire notre opération. Il y a donc une erreur de syntaxe qui doit être signalée dans notre fonction principale.

9. A tout moment, si un chiffre n'a pas la place d'être alloué, qu'un push a échoué, ou que toute autre opération d'allocation mémoire a échoué, on retourne un `NULL` qui sera traité dans la fonction d'évaluation postfixe. La fonction d'évaluation postfixe va par la suite signaler à la fonction main du programme qu'il n'y a plus d'espace suffisant pour effectuer l'évaluation de l'expression. Dans ces cas-là, on vide la pile et le buffer.

Afin que la mémoire soit en tout temps protégée des problèmes de mémoire, un buffer sera utilisé afin de stocker les variables actuellement affectées dans l'expression.

2. Résolution des problèmes

a) Représentation des variables et des nombres.

```
struct num { int compteurRef; int negatif; struct cell *chiffres; };
struct cell{ char chiffre; struct cell *suivant; };
```

Un nombre est représenté par un compteur de référence, par son signe (0 si positif, 1 si négatif) et a un pointeur sur son chiffre de poids le plus faible qui le compose.

Les chiffres sont représentés par la structure `cell`. On ordonne les nombres comme étant des listes chaînés de chiffre du poids le plus faible au plus fort.

Un chiffre est sous forme de `char`, et a un pointeur sur le chiffre suivant qui le suit par ordre de poids dans le nombre.

```
struct memoire {struct variable *tete;};
struct variable{char var; struct num *nombre; struct variable *suivant;};
```

Dans la même idée, nous avons créé les variables et la mémoire sous forme de liste chaînée. La structure mémoire contient un pointeur sur la première variable qui a été stockée dans celle-ci.

Les variables contiennent: un **`char`** `var` qui est la représentation sous forme de caractère de la variable; un pointeur sur le nombre auquel est associée la variable; et un pointeur sur la variable suivante qui a été stockée après elle.

b) Analyse de chaque ligne et calcul de la réponse

```
struct node {num *nombre; struct node *suivant;};
struct pile {int length; struct node *top;};
```

On récupère les lignes avec la fonction `getchar()` dans une boucle qui ne s'arrête que lorsque que `getchar()==EOF`. A chaque début de ligne, on alloue de l'espace pour une pile et un buffer qui contiendra les variables affectées durant l'évaluation de l'expression postfixe afin de protéger la mémoire principale.

Dans la boucle `getchar()`, dès que l'on repère un mot (un mot est séparé par exactement deux espaces, sauf au début de la ligne et à la fin de la ligne), celui-ci est évalué par la fonction `postfixeEvaluation` qui en vérifie la validité. Si ce mot est valide, la fonction va avoir plusieurs options disponibles:

- si c'est un **nombre entier**: elle appelle une fonction `transformationStructure` qui transforme le mot en nombre. La fonction vérifie que ce nombre n'existe pas déjà en mémoire, dans le buffer ou dans la pile. Si il existe déjà, alors la fonction retourne un pointeur sur ce nombre. Sinon elle retourne un nouveau nombre avec un compteur de référence initialisé à 0. Ce pointeur de nombre est alors poussé dans la pile, ce qui incrémente son compteur de référence.

- si c'est une **variable**: `postfixeEvaluation` va chercher un pointeur du nombre dans le buffer ou dans la mémoire principale en privilégiant la valeur de la variable contenue dans le buffer qui est la plus "actuelle". Puis le nombre retourné est poussé dans la pile.

- si c'est un **opérateur binaire**: on pop deux valeurs de la pile et on effectue l'opération demandée. On récupère le résultat. Encore une fois, on vérifie que ce nombre n'existe pas déjà en mémoire, dans le buffer ou dans la pile. Si oui, on retrouve un pointeur sur ce nombre déjà présent, et on le pousse dans la pile. La difficulté dans ce cas-ci est la suivante: les valeurs "popés" précédemment peuvent voir leur compteur de référence à 0 mais sont égales. Afin de ne pas les désallouer deux fois, on vérifie si elles ne sont pas égales entre elles afin de ne désallouer qu'une seule fois la valeur. Si elles ne sont pas égales, on peut désallouer les deux valeurs dont le compteur de référence est tombé à 0.

- si c'est un **opérateur unaire**: si on veut obtenir le compteur de référence d'un nombre, on récupère le compteur de référence du nombre que l'on va transformer en nouveau nombre en appelant `transformationStructure`. Un pointeur vers le nouveau nombre (représentant donc le compteur de référence) sera retourné par la fonction. Si on fait une affectation, il suffit alors d'affecter la variable avec le pointeur de nombre dans le buffer et incrémenter le compteur de référence.

On répète l'appel à `postfixeEvaluation` à chaque nouveau mot trouvé sur la ligne jusqu'à atteindre `'\n'`. À ce moment, si la pile n'a qu'une seule valeur, l'expression postfixe est valide et on peut imprimer le résultat dans la console. On actualise les valeurs des variables contenues en mémoire grâce au buffer qui contenait toutes les variables affectées durant l'évaluation de l'expression. On vide la pile et le buffer. Et enfin on peut recommencer l'analyse d'une nouvelle ligne.

---

c) Gestion de la mémoire

```
struct memoire {struct variable *tete;};
struct variable{char var; struct num *nombre; struct variable *suivant;};
```

La mémoire principale, qui contient les variables affectées, sera initialisée dès le lancement du programme.

Au moment de l'évaluation d'une ligne, on créé un buffer qui a une structure identique à celle de la mémoire. Ce buffer va contenir les variables qui sont affectées durant le processus d'évaluation.

Si on fait une affectation, on effectue l'affectation de la variable dans un buffer. En effet, si une erreur vient se glisser dans l'expression postfixe, la mémoire principale ne sera pas corrompue par les valeurs affectées sur la ligne contenant l'erreur.

Si on fait appel à une variable lors de l'évaluation d'une expression, on va d'abord chercher dans le buffer afin de retourner la valeur la plus actuelle associée à cette variable. Si cette variable n'existe pas dans le buffer, on cherche dans la mémoire principale et on retourne un pointeur sur ce nombre.

S'il n'y a pas eu d'erreurs lors de l'évaluation de l'expression postfixe, on peut actualiser les valeurs contenues dans la mémoire grâce au buffer qui contient les nouvelles valeurs affectées, et on alloue éventuellement des blocs mémoires supplémentaires pour les nouvelles

variables non-affectées (on fait donc une vérification préalable que les appels à `malloc` engendrés par le rétablissement des valeurs ne retournera pas `NULL` dans cette opération).
Le buffer est vidé à chaque analyse de nouvelle ligne. La mémoire principale reste intacte.

---

d) Algorithmes d'addition, soustraction et multiplication

On a implémenté les algorithmes d'addition, de soustraction et de multiplication de manière assez classique.

Pour chaque opération, on traite d'abord les cas où 0 est une opérande. Dans le cas de l'addition et de la soustraction, il suffit de renvoyer une copie du nombre qui n'est 0 comme résultat. Dans le cas de la multiplication, il suffit de renvoyer 0.

Puis, on regarde le signe de chaque opérande:

- dans le cas de l'**addition**, si les deux opérandes sont négatives, alors le résultat de la somme de deux entiers sera négatif. Si une des deux opérandes sont négatives, on effectue une soustraction. On copie le résultat de cette soustraction et on retourne le résultat l'addition.

- dans le cas de la **soustraction**, si c'est une soustraction d'un nombre positif avec un nombre négatif par exemple, il suffit de récupérer le résultat de l'addition de ces deux nombres. Si c'est la soustraction d'un nombre négatif avec un négatif, on récupère le résultat de la soustraction en inversant l'ordre des opérandes, etc.

- dans le cas de la **multiplication**, en fonction du signe des opérandes, on change le signe du résultat.

La boucle qui effectue réellement le calcul dans l'addition permet de faire l'addition de deux entiers positifs. S'il y a un reste après l'addition de tous les chiffres, on ajoute ce chiffre à la fin du nouveau nombre.

La boucle permettant de faire la soustraction de deux nombres entiers est possible seulement si la valeur soustraite est plus petite que la valeur que l'on veut soustraire (p.e $11 - 12$). Dans ce cas-là, on fait un nouvel appel à la soustraction en effectuant $12 - 11$ et on donne un signe négatif au résultat de cette soustraction afin d'obtenir le bon résultat de la soustraction $11 - 12$. Les poids forts résiduels sont enlevés grâce à une fonction qui enlève tous les 0 en position de poids le plus fort.

Dans le cas de la multiplication, on s'assure seulement que la première opérande est plus longue en nombre de chiffres que la seconde opérande. Sinon on fait un appel à multiplication en inversant les opérandes. Puis on effectue des additions successives. Par exemple, pour $17 * 3$, on effectue 7 fois l'addition de 3. 21 est alors ajouté au résultat. On effectue ensuite l'addition de 3 une fois. On récupère 3. Afin d'éviter de très longues boucles inutiles, on ajoute un 0 à ce nouveau résultat, ce qui donne 30. Puis on additionne 30 à 21, ce qui donne 51. Et on retourne le résultat puisque que nous avons parcouru toute l'opérande 17.

e) Traitement des erreurs

Si une erreur d'allocation mémoire apparait au moment de l'addition, de la soustraction ou de la multiplication (1 est la valeur de retour d'un push, de l'ajout d'un chiffre, etc. lorsqu'un problème d'allocation a eu lieu), celles-ci renverront `NULL` à la fonction d'évaluation `postfixeEvaluation`.

Ce `NULL` sera interprété comme une erreur d'allocation mémoire, et donc `postfixeEvaluation` renverra la valeur 1 à la fonction principale qui comprendra qu'un problème d'allocation mémoire s'est produit. Dans ces cas-là, la variable `exception` présente dans la fonction principale prend la valeur 1 et permet d'afficher le message d'erreur : Out of memory. Puis le buffer, ainsi que la pile seront détruites et on recommence à la nouvelle ligne.

Les erreurs de type syntaxe sont gérées par par les fonctions `validLiteral`, `validopBin` et `validOpUn` qui permettent de valider la syntaxe d'un mot. Si ces fonctions retournent tous faux dans la fonction d'évaluation postfixe, `postfixeEvaluation` retournera 2 (syntaxe incorrecte de l'expression). Également, si une variable non-affectée apparaît sur une ligne et que celle-ci n'est pas affectée, ni dans le buffer, ni en mémoire, `postfixeEvaluation` retournera 2. Ainsi, la variable `exception` prendra la valeur 2, et une erreur de syntaxe sera signalée.

De plus, si tout s'est bien passé durant l'évaluation de l'expression, mais que la pile ne contient pas exactement un élément au moment de l'affichage du résultat, c'est qu'il y a une erreur de syntaxe, donc `exception` prendra la valeur 2. Puis le buffer ainsi que la pile seront détruites.