RAPPORT – TRAVAIL PRATIQUE

Cours: IFT2035

TP numéro : 1

Date de remise: 17 octobre 2016

Noms: Farzin Faridfar

Catherine Laprise

1. Fonctionnement général du programme

Nous avons utilisé un systèmes de piles avec des listes chaînées de structs pour représenter la calculatrice.

Tout d'abord les entiers sont représentés à l'aide d'un struct de type *num*, qui contient entre autre le signe de l'entier, et qui pointe vers une pile de structs de type *cell*. Chaque chiffre de l'entier est stocké dans un *cell* séparé.

Après la lecture d'un entier ou d'une variable, on créé un struct de type *operande*, qui contient un pointeur vers le *num* associé à l'entier. On empile ensuite la nouvelle opérande dans la pile *pile_op*, à l'aide de la méthode *pushOp*. On a donc pendant tout le temps du calcul de la ligne un accès vers les entiers lus, le plus récent étant à la tête de la pile (LIFO).

Lors de la lecture d'un opérateur, on effectue donc l'opération demandée sur les opérandes stockés dans la pile. Les opérateurs binaires (+, -, *) s'effectuent sur les deux premiers éléments de la pile, alors que les opérateurs unaires ? et = utilisent l'opérande en tête de file. Après chaque opération, le ou les opérandes traités sont retirés de la pile et remplacés par le résultat de l''opération. On répète ce processus jusqu'à la fin de la lecture de la ligne, après quoi il ne devrait rester qu'un seul opérande sur la pile, qui représente le résultat final. Une pile finale vide ou contenant plusieurs opérandes engendre une erreur.

2. Problèmes de programmation

(a) Comment les nombres et variables sont représentés

Chaque nombre est représenté par un struct de type *num* qui contient le signe du nombre, un compteur de référence, ainsi qu'un pointeur vers un *cell*. Chaque chiffre du nombre est stocké dans un *cell* et contient un pointeur *suivant* ; on peut donc empiler l'entier chiffre par chiffre au moyen de la méthode *pushCellFront*. Une fois terminé, le pointeur *num.chiffres* pointe vers le *cell* contenant le chiffre de poids le plus faible. En ce qui concerne le nombre 0, celui-ci n'est jamais représenté s'il est en début de nombre. De plus, le nombre 0 est représenté par un pointeur null dans *num.chiffres* pour alléger les calculs.

Il est à noter que les chiffres sont stockés dans un type *char ;* on doit donc faire la conversion lors des calculs pour le transformer en la bonne valeur de type *int* (-48).

Le domaine des variables étant fini et relativement petit, nous avons fait le choix de les gérer à l'aide d'un tableau de taille fixe initialisé au début du programme. Cela nous a évité la gestion de mémoire reliée aux piles et nous permet également un accès direct à chaque variable via un index pour une meilleure performance. Pour éviter les dédoublements, le tableau contient uniquement des pointeurs vers des *num*, nous évitant ainsi d'avoir à créer des copies des chiffres. Lors de l'assignation, on pointe simplement vers le *num* voulu dans la pile d'opérandes et on augmente son compteur de référence. On peut ainsi conserver la référence vers le *num* même si on retire l'opérande de la pile.

Lors du remplacement ou de la suppression d'une variable déjà initialisée, on décrémente son compteur de référence. Le *num* est alors libéré de la mémoire seulement si son compteur est à 0.

(b) Comment se fait l'analyse de chaque ligne et le calcul de la réponse

Les lignes sont lues un caractère à la fois au moyen de la fonction *getchar*. La fonction *lireChiffres* est responsable du traitement des chiffres lus. Pour la lecture des entiers, un *int « continue_nb »* permet d'indiquer si le chiffre lu fait partie d'un nombre en cours de lecture ou s'il s'agit du premier chiffre d'un nombre (dans quel cas un nouvel *operande* et un nouveau *num* sont créés et empilés sur *pile_op* avec la fonction *pushOp).* Comme mentionné plus haut, les chiffres sont stockés un à un dans des *cell* qui sont empilés à partir de *num.chiffres.* La lecture d'un espace sert de délimiteur entre les chiffres et entre les opérateurs et remet *continue_nb* à 0.

Lors de la lecture des caractères alphabétiques représentant des variables, on empile également l'entier associé à cette variable sur *pileOp* afin que la variable puisse être utilisée dans une opération au même titre que les autres entiers. Le nouvel opérande pointe sur le même *num* que le tableau de variables afin d'éviter toute copie.

Les opérateurs sont traités par la fonction *lireOperateurs*, qui appelle les fonctions appropriées selon l'opérateur détecté. Les opérateurs binaires (+, -, *) font d'abord l'objet d'une vérification des signes de chaque opérande.

Les opérations sont réalisées à partir de la pile d'opérandes *pile_op*. Une fois l'algorithme effectué, les opérandes sont supprimées de la pile et le résultat y est empilé. Ainsi, supposant un nombre d'opérandes et d'opérateurs adéquats et une bonne syntaxe, à la fin de la lecture de la ligne, la fonction *printResultat* devrait trouver une seule opérande restante dans la pile, représentant le résultat final.

(c) Comment se fait la gestion de la mémoire

La création de tous les nouveaux struct se fait par appel à *malloc*, passant en paramètre le type de l'objet désiré. Les fonctions *suppOp* et *suppNum* se chargent de libérer l'espace alloué aux *operande* et aux *num* respectivement. *SuppOp* libère l'opérande se trouvant en tête de pile (« *pop* »), prenant soin de tout d'abord décrémenter le compteur de référence du *num* associé et d'appeler *suppNum* sur celui-ci si nécessaire. *SuppNum* commence par supprimer tous les *cell* empilés à partir de *num.chiffres* pour éviter toute fuite de mémoire, et supprime le *num* avant de se terminer.

À la fin de la lecture de chaque ligne, la pile est vidée complètement par la fonction *videPileOp*. Seuls les *nums* dont le compteur de référence n'est pas à 0 (donc stockées dans une variable) sont conservés. De même à la fin du programme, on vide entièrement la pile avec *videPileOp* ainsi que le tableau de variables via la fonction *videVariables.* On vérifie en tout temps le compteur de référence des *num* traités afin d'éviter de libérer des *num* en cours d'utilisation, ou de libérer plusieurs fois la même variable.

(d) Comment les algorithmes d'addition, soustraction et multiplication sont implantés

Comme mentionné précédemment, les opérations se font directement à partir des opérandes placées sur la pile *pile_op*. On ne passe donc aucun élément en paramètre.

Une vérification des signes est d'abord effectuée pour simplifier les algorithmes (fonctions *lireSignesAdd, lireSignesSoust et lireSignesMult)*. En effet, les opérations se font directement sur les *cell*, donc en valeur absolue. Le signe est traité après le calcul du résultat. Par exemple, la soustraction (-3) 2 - sera traitée comme une addition suivi d'un changement de signe, - (3 2 +) pour alléger les calculs.

- Fonction *add*: La fonction vérifie tout d'abord si un des opérandes est 0. Si c'est le cas, on conservera uniquement l'opérande pertinent sans faire de calcul. Si les opérandes sont non nuls, la fonction fait la somme de chacun des chiffres des nombre – un par un – en commençant par le chiffre de poids faible, auquel on ajoute le reste d'addition des chiffres précédents si

nécessaire.. On conserve donc à chaque étape des variables intermédiaires représentant la somme des chiffres, le reste d'addition pour les valeurs > 10 et le modulo 10 de la somme. On prend le modulo de chaque résultat et on l'ajoute à un nouveau *num* « *resultat »* créé à cet effet. Comme cette fois on traite les chiffres de poids faibles en premier, on utilise la méthode *pushCellBack* qui ajoute les nouveaux *cell* en fin de liste (FIFO) afin que le pointeur *num.chiffres* pointe toujours vers le chiffre de poids le plus faible. À la fin de l'algorithme, on ajoute le reste une dernière fois au besoin, on supprime les opérandes utilisés et on ajoute *resultat* sur *pile_op*.

- *Fonction soust* : Pour cette opération on doit tout d'abord vérifier si un des opérandes est plus grand que l'autre afin de simplifier l'algorithme. On parcoure donc les deux opérandes pour identifier le plus grand, puis on fait le calcul en prenant le plus grand moins le plus petit. On ajuste finalement le signe à la fin de l'opération.
- *Fonction mult*: Ici, on fait exactement ce qu'on fait sur un papier. Tout d'abord, dans le cas où un des deux opérandes est null, on retourne directement un résultat null sans faire le calcul. Sinon, on multiplie chaque chiffre du 2^{e} opérande à tous les chiffres du 1er opérande, utilisant les valeurs de reste et de modulo, de manière semblable à l'addition. On stocke chaque résultat intermédiaire sur *pile_op.* Une fois les mutiplications complétées, on appelle la fonction *add* afin de compiler le résultat final. Finalement, on supprime les opérandes originales.

(e) Comment se fait le traitement des erreurs

Plusieurs fonctions de validation ont été ajoutées afin de gérer les erreurs.

Lors de la lecture, les fonctions *detectEntier, detectAlpha* et *detectOp* vérifie que les caractères entrées sur la ligne de commande sont bien valides (variables entre a et z, opérateurs valides, etc.). On fait également une vérification des espaces entre les caractères. Les messages d'erreurs donnent des indications sur l'usage et la syntaxe correcte.

Chaque appel à *malloc* est également suivi d'une vérification que la valeur retournée n'est pas *null*. En cas de manque de mémoire, le programme est arrêté.

Lorsqu'on utilise la pile *pile_op*, le programme vérifie également que la pile n'est pas vide est affiche immédiatement un message d'erreur si une valeur attendue est nulle.

Comme mentionné à la section sur la gestion de la mémoire, des fonctions de nettoyage et de libération de la mémoire allouée sont appelées pour éviter les pointeurs fous ou les fuites de mémoire. Notamment les struct *num* font toujours l'objet d'une vérification de leur compteur avant d'être libérés.

Toutes les fonctions de suppression des différents structs vérifient tout la « hiérarchie » de chaque objet (*num*, liste de *cell)* pour s'assurer de libérer toute la mémoire inutilisée.