Algorithme de shunting-yard

Rémi AIRIAU

22204545 – Licence Informatique – INXPA31B11
Enseignant de TP : SMAUS JAN-GEORG
Séance 2

12/01/2024

UE Algorithmique 3

Résumé

Ce compte-rendu porte sur la séquence numéro 2 des travaux pratiques de l'UE Algorithmique 3. Tout d'abord, j'expliquerai mes choix d'implantations pour les différentes fonctions et algorithmes développés par rapport aux performances et à la robustesse. Ensuite, je décrirai le comportement du programme sur les jeux de tests fournis. De plus, je présenterai les quelques améliorations proposées et utilisées afin de répondre à la problématique. Enfin, je donnerai mon analyse personnelle avec, pour chaque partie, les problèmes rencontrés et les solutions mises en place.

Mots-clefs

Analyse lexicale ; Token ; Files ; Piles ; Notation infixe ; Notation prefix ; Shunting-Yard ; Évaluation d'expressions arithmétiques ;

Table des matières

I – Implémentations

I.1 Fonction stringToTokenQueue

L'implémentation de cette fonction est presque la traduction directe de l'algorithme proposé dans le sujet. Cependant, avec une simple modification, elle permet dans ma version d'être plus robuste et de traiter n'importe quelle chaîne de caractères récupérée à partir de la ligne d'un fichier. En effet, les données du fichier pourraient avoir été involontairement modifiées et des caractères invisibles ou bien spéciaux pourraient se glisser dans l'expression. Pour cela, je ne décale pas seulement le curseur à chaque fois qu'il rencontre un espace ou un retour à la ligne, mais dès qu'il rencontre un caractère différent d'un symbole supporté par le programme ou bien différent d'un caractère alpha-numérique. J'ai alors rajouté la fonction suivante qui teste si le caractère peut être directement converti en chiffre :

```
1
2 bool isNumValue(char c) {
3     return 48 <= c && c <= 57;
4 }
```

Elle a le même rôle que la fonction isdigit(), mais je me suis interdit de modifier les inclusions d'en-têtes. Le deuxième choix important est l'utilisation d'un pointeur temporaire pour compter le nombre de caractères composant un nombre. Si on utilise directement le curseur principal, on perdra l'adresse de début au moment de créer le token.

```
1
2 /* curpos pointe sur une valeur numerique */
3 valueLength = 0; /* Initisalisation du compteur a 0 */
4 numberPtr = curpos; /* Recherche de la fin du nombre a partir du curseur */
5 while (isNumValue(*numberPtr) || *numberPtr == '.') {
6     /* Si le caractere est un chiffre ou un nombre flottant, on incremente la longueur
      et on passe au suivant */
7     valueLength++;
8     numberPtr++;
9 }
10 token = createTokenFromString(curpos, valueLength);
```

I.2 Fonction shuntingYard

Là encore, l'implémentation suit exactement le pseudo-code fourni dans le sujet. Pour éviter la redondance de code, j'ai rajouté trois fonctions auxiliaires :

```
1     Token* getTopAndPop(Stack* s);
2     bool tokenIsLeftParenthesis(Token *t);
3     bool tokenIsRightParenthesis(Token *t);
```

Ainsi, le code de la fonction *shuntingYard* est plus claire (notamment pour les conditions) ce qui permet d'éviter de mettre trop de commentaires. La compléxité en temps de cet algorithme est de l'ordre de $O(n)$, où n est la taille de la file en notation infixe en entrée.

I.3 Fonction evaluateExpression

Le pseudo-code de cette fonction est fournie. Lors de l'implémentation en langage C, je réutilise ma fonction auxiliaire getTopAndPop pour récupérer les deux opérandes en tête de pile et évaluer l'expression avec le token opérateur. Il faut alors libérer les deux opérandes ainsi que le token. À la fin du traitement, j'utilise un pointeur pour récupérer le résultat et libérer les dernières ressources. La compléxité en temps de cet algorithme est également de l'ordre de $O(n)$, n étant la taille de la file en notation postfixe.

I.4 Fonction computeExpression

Pour cette fonction, j'ai décidé de gérer le cas d'erreur où l'espace mémoire réservé au buffer qui contiendra les expressions ne peut pas être alloué. Il s'agit simplement d'une bonne pratique :

```
if (buffer == NULL) {
    perror("Allocation for buffer has failed.");
    exit(MALLOC_ERROR);
}
```

Aussi, si l'expression contient moins de deux caractères, je choisis de ne pas la traiter. En effet, même si je n'ai pas ajouté cette fonctionnalité, on pourrait faire en sorte que le programme supporte l'utilisation de la factorielle : $n!$

Enfin, il faut libérer les deux files utilisées par cette fonction et qui permettent de faire le lien entre les différents algorithmes.
À noter qu'il y a deux endroits où l'on peut libérer ces ressources :

- Soit à l'intérieur de la fonction qui prend la file en paramètre et qui la vide au fur et à mesure de l'exécution de l'algorithme si l'on ne copie pas le pointeur. Dans ce cas, *shuntingYard* vide la file en notation infixe pour remplir celle en notation postfixe, *evaluateExpression* quant à elle vide la file en notation postfixe pour retourner une valeur.
- Soit dans la fonction *computeExpression*. Si l'on effectue une copie du pointeur au début de chaque fonction, il faudra utiliser $freeTokenQueue$ III.3.2, sinon *deleteQueue* suffit.

J'ai fait le choix de libérer les ressources restantes dans *compteExpression* afin de centraliser à la fois l'affichage et la gestion de la mémoire dans une même fonction principale qui est par ailleurs la seule appelée dans le *main*. On met à la fin :

```
deleteQueue(&queueInfix);
deleteQueue(&queuePostfix);
```

II – Tests

Sur le fichier de tests fourni, le comportement est celui attendu et toutes les ressources mémoires sont libérées correctement comme le montre l'affichage du terminal ci-dessous :

```
~/Documents/Github/Algo3/TP/TP2/Code$ valgrind --track-origins=yes
--leak-check=full --leak-resolution=high ./expr_ex1 ../Test/exercice1.txt
==6055== Memcheck, a memory error detector
==6055== Copyright (C) 2002-2017, and GNU GPL d, by Julian Seward et al.
==6055== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6055== Command: ./expr_ex1 ../Test/exercice1.txt
==6055==
Input     : 1 + 2 * 3
Infix     : (5) --  1.000000 + 2.000000 * 3.000000
Postfix   : (5) --  1.000000 2.000000 3.000000 * +
Evaluate  : 7.000000

Input     : (1+2) * 3
Infix     : (7) --  ( 1.000000 + 2.000000 ) * 3.000000
Postfix   : (5) --  1.000000 2.000000 + 3.000000 *
Evaluate  : 9.000000

Input     : 1+2^3*4
Infix     : (7) --  1.000000 + 2.000000 ^ 3.000000 * 4.000000
Postfix   : (7) --  1.000000 2.000000 3.000000 ^ 4.000000 * +
Evaluate  : 33.000000

Input     : (1+2)^3*4
Infix     : (9) --  ( 1.000000 + 2.000000 ) ^ 3.000000 * 4.000000
Postfix   : (7) --  1.000000 2.000000 + 3.000000 ^ 4.000000 *
Evaluate  : 108.000000

Input     : 1+2^(3*4)
Infix     : (9) --  1.000000 + 2.000000 ^ ( 3.000000 * 4.000000 )
Postfix   : (7) --  1.000000 2.000000 3.000000 4.000000 * ^ +
Evaluate  : 4097.000000

Input     :  1 + 2^3 * 4 * 5
Infix     : (9) --  1.000000 + 2.000000 ^ 3.000000 * 4.000000 * 5.000000
Postfix   : (9) --  1.000000 2.000000 3.000000 ^ 4.000000 * 5.000000 * +
Evaluate  : 161.000000

Input     : (1 + 2^(3 * 4)) * 5
Infix     : (13) --  ( 1.000000 + 2.000000 ^ ( 3.000000 * 4.000000 ) ) * 5.000000
Postfix   : (9) --  1.000000 2.000000 3.000000 4.000000 * ^ + 5.000000 *
Evaluate  : 20485.000000

==6055==
==6055== HEAP SUMMARY:
==6055==     in use at exit: 0 bytes in 0 blocks
==6055==   total heap usage: 220 allocs, 220 frees, 9,416 bytes allocated
==6055==
==6055== All heap blocks were freed -- no leaks are possible
==6055==
==6055== For lists of detected and suppressed errors, rerun with: -s
==6055== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Étant donné que la fonction *stringToToken* est assez robuste, on peut même modifier le fichier de tests en insérant des caractères quelconques, par exemple comme ceci (le même affichage est produit) :

```
1 + 2 * 3()
(1+2) * 3
1+2^3*4
(1+2)^3*4
1+d$2^lkf (f3*4s)

 1 +g 2^h3 * agb4 * !5
(1 + 2^(3 *df 4)) * 5
```

III – Analyse personnelle

III.1 Améliorations possibles

III.1.1 Comportement sur les tests

Comme détaillé dans la section précédente, le comportement du programme sur les tests fournis est satisfaisant. Cependant, des bugs de mémoire commencent à apparaître, par exemple si l'on écrit tout simplement du texte, des paragraphes donc, sans valeurs numériques ni opérateurs. Il faut tout de même préciser que le programme se termine normalement en affichant encore une fois le résultat attendu, sans leaks de mémoire. Voici certaines des erreurs affichées par valgrind uniquement :

```
...
==6459==  Uninitialised value was created by a heap allocation
...
==6459== Conditional jump or move depends on uninitialised value(s)
...
==6459== Syscall param write(buf) points to uninitialised byte(s)
...
```

On pourrait peut-être régler ce problème en rajoutant une variable qui compte le nombre d'opérateurs et de valeurs trouvées dans la chaîne. Si ce nombre est au moins de 3 (sans opérateurs unaires) on alloue l'espace nécessaire pour les tokens. De plus, il ne faut pas "mal parenthéser" l'expression en rajoutant aléatoirement des parenthèses à la fin. Dans ce cas, on pourrait faire en sorte que le programme ait une manière "par défaut" d'interpréter une expression arithmétique sous forme de chaîne de caractères.

III.1.2 Opérations supportées

On pourrait aussi ajouter la gestion d'opérateurs unaires permettant par exemple de gérer la factorielle ou encore les nombres négatifs. Dans ce dernier cas, il faudrait réfléchir sur comment les différencier de la soustraction. Si l'on rajoute des opérateurs, il faut modifier les fichiers token.c et token.h pour redéfinir les priorités et les symboles supportés. Et il faudrait aussi modifier les fonctions écrites dans ce TP en conséquence.

III.2 Problèmes rencontrés

Les problèmes rencontrés étaient systématiquement des problèmes de mémoire. Autrement, si le programme ne s'arrêtait pas suite à un *SegmentationFault*, l'affichage et le résultat étaient souvent correctes.

La principale difficulté concernait la fonction *shuntingYard*. Au premier abord, il était assez difficile, à l'aide du pseudo-code dans le sujet et celui sur la page wikipédia, de comprendre l'algorithme et de ne pas se tromper sur la priorité et l'ordre des conditions, par exemple :

```
1 while (!stackEmpty(opStack)
2         && (tokenGetOperatorPriority((Token*)stackTop(opStack)) >
      tokenGetOperatorPriority(token)
3         || (tokenGetOperatorPriority((Token*)stackTop(opStack)) ==
      tokenGetOperatorPriority(token)
4         && tokenOperatorIsLeftAssociative(token)
5         && !tokenIsLeftParenthesis((Token*)stackTop(opStack)))))
6 {
7         postfix = queuePush(postfix, getTopAndPop(opStack));
8 }
9 ...
```

Ensuite, contrairement aux fonctions *evaluateExpression* et *computeExpressions* dans lesquelles il est facile de savoir quand libérer de la mémoire, j'ai trouvé que cela était beaucoup moins évident pour *shuntingYard*.

Enfin, plus tôt dans le déroulement du TP, je me suis également quelque peu heurté à la question 3 de la première partie. À savoir : comment libérer la file de tokens obtenue après exécution de la fonction *stringToTokenQueue*, et ce dans la fonction *computeExpression* ?

III.3 Solutions

III.3.1 Libération mémoire dans shuntingYard

Finalement, après avoir analysé le code et avec de la patience, j'ai fini par trouver un cas dans lequel il faut libérer des tokens. En effet, si l'on trouve dans la file en notation infixe une parenthèse droite, il faut bien penser à libérer la mémoire occupée par ce token, ainsi que celle occupée par la parenthèse gauche correspondante car ces deux symboles ne seront plus utilisés dans la notation postfixe.

```
if (!stackEmpty(opStack)) {
    /* Si la pile n'est pas vide, le top contient une parenthese gauche */
    Token* toDelete = (Token*)stackTop(opStack); /* Recuperation de la parenthese
    gauche a supprimer */
    opStack = stackPop(opStack);
    /* Liberation des tokens (parentheses) qui ne seront plus utilises */
    deleteToken(&toDelete);
    deleteToken(&token);
}
```

III.3.2 Libérer une file de Tokens

Afin d'avoir un bilan de mémoire nul à la fin de la fonction *computeExpression*, j'ai écrit une fonction

```
void freeTokenQueue(ptrQueue* q);
```

qui permet de libérer toutes les ressources (tokens) allouées à l'extérieur du gestionnaire de collection (la file). Afin de respecter les propriétés de l'interface privée, on utilise seulement les fonctions fournies par l'interface publique du gestionnaire de collection (ici le module Queue composé des fichiers queue.c et queue.h). Ainsi, tant que la file n'est pas vide, on récupère l'élément en tête (*queueTop*), qu'on libère avec *deleteToken* et puis on supprime (*queuePop*) l'élément de la file. À la fin, on libère cette dernière. Dans la fonction *computeExpressions*, on appelle alors $freeTokenQueue$ lorsque le traitement sur une ligne est terminé.
Dans le code, cette fonction n'est plus utilisée à partir de la seconde partie du sujet. En effet, *stringToTokenQueue* est la seule fonction qui alloue des tokens (à l'aide de *createTokenFromString* du module Token) suite au traitement d'une chaîne de caractères. Les autres algorithmes (*shuntingYard* et *evaluateExpression*) ne font que les libérer car je ne copie pas les pointeurs en début de fonction (la file est directement modifiée). I.4
Pour résumer, cela signifie que toutes les ressources allouées à l'extérieur du gestionnaire de collection ont déjà été libérées naturellement par les algorithmes. Il suffit d'utiliser la fonction *deleteQueue* pour libérer les files utilisées.