

Algorithme de Shunting-yard.

Algorithmique 3 - Travaux pratiques

Les travaux pratiques seront réalisés à partir de l'archive `base_code_lab2.tar.gz` fournie sur moodle et contenant la hiérarchie suivante :

`base_code_lab2`

- > **Test** : répertoire contenant les fichiers de test.
- > **Code** : répertoire contenant le code source fourni devant être complété.

Il est demandé à l'étudiant de rédiger, dans un fichier au format **pdf**, un compte rendu de travaux pratiques contenant, pour chaque exercice :

- les réponses aux éventuelles questions posées pour l'exercice dans le sujet,
- une description synthétique des algorithmes mis en œuvre et une analyse de la complexité en temps et en espace de ces algorithmes,
- une synthèse de la démarche de programmation mise en œuvre, en précisant :
 - les références concernant les documentations et aides utilisées (livres, camarades, chat-GPT, site web ...),
 - les difficultés rencontrées et comment elles ont été résolues.

À l'issue de la séquence, l'étudiant devra déposer sur moodle une archive selon les spécifications suivantes :

- Format de l'archive identique au format initial (`.tar.gz`),
- Contenu de l'archive :
 - le compte rendu de TP au format pdf,
 - un dossier **Code** contenant uniquement le code source et le Makefile, à l'exclusion de tout autre fichier,
 - un dossier **Test** contenant les tests fournis et éventuellement augmentés de nouveaux cas de test.

Le respect strict des consignes ci-dessus est un élément de l'évaluation des travaux déposés.

L'objectif de cette séquence est d'écrire un programme permettant d'évaluer un ensemble d'expressions arithmétiques fournies sous la forme de chaîne de caractères en notation infixe par application de l'Algorithme de Shunting-Yard pour la transformation de la notation infixe vers la notation post-fixe. Cette séquence positionne l'étudiant en tant qu'utilisateur de modules C implantant les types abstraits de données pile, file et élément syntaxique d'expressions arithmétiques.

Les expressions arithmétiques pouvant être évaluées, sont construites avec les éléments suivants :

- Valeurs numériques réelles ne contenant que des nombres *positifs*.
- Opérateurs binaires `+`, `-`, `*`, `/`, `^` représentant respectivement l'addition, la soustraction, la multiplication et la fonction puissance de deux nombres réels. Ces opérateurs respectent les propriétés de priorité et d'associativité telles que définies en arithmétique classique.
- Parenthèses permettant de modifier la priorité des opérateurs.
- Les éléments d'une expression peuvent être ou ne pas être séparés par des espaces.

Les expressions suivantes, contenues dans le fichier de test `Test/exemple1.txt` sont des expressions arithmétiques exploitables par le programme que l'on souhaite écrire, dont l'évaluation

donne des valeurs bien différentes :

```
1 + 2 * 3
(1+2) * 3
1+2^3*4
(1+2)^3*4
1+2^(3*4)
1 + 2^3 * 4 * 5
(1 + 2^(3 * 4)) * 5
```

En utilisant les implantations des types abstraits de données fournis ([Token](#), [Stack](#), [Queue](#)) décrits ci-dessous, vous devez réaliser les 3 exercices suivants.

Les types abstraits [Stack](#) et [Queue](#) correspondent à l'implantation statique de la pile et dynamique de la file, telles qu'elles ont été vues en cours-TD. Ces implantations sont cependant rendues indépendantes du type des informations stockées, en proposant une gestion de collection de pointeurs non typés ([void *](#)).

1 Analyse lexicale et découpage en *Token* d'une expression arithmétique

Concepts étudiés :

- Éléments syntaxiques des expressions arithmétiques.
- Utilisation de collections abstraites et génériques.
- Notion de propriété sur les données.

Compétences acquises dans cette partie :

- Savoir lire et analyser une chaîne de caractères depuis un fichier.
- Savoir exploiter une interface pour développer un traitement des données.

Lorsque l'on souhaite traduire une expression arithmétique (ou plus généralement le code source d'un programme) de sa représentation naturelle vers une représentation exploitable pour le calcul (ou plus généralement vers du code binaire exploitable par le processeur), la première étape consiste en l'analyse lexicale de cette représentation naturelle (ou code source) pour générer un ensemble d'éléments syntaxiques, nommés *Token*.

Dans cet exercice, vous devez programmer un ensemble de fonctions permettant de convertir une chaîne de caractères représentant l'expression à analyser en une file de *Token*.

Le module `Token.h/Token.c` fournit l'ensemble des fonctions permettant de construire, détruire et exploiter un *Token*. Le fichier `Token.h` contient en commentaire la spécification de ce TAD qui est aussi consultable dans la documentation générée par doxygen.

Le fichier `main.c` contient le squelette du programme principal. C'est dans ce fichier que vous devez écrire les fonctions demandées ci-dessous.

1.1 Lecture des chaînes de caractères contenues dans le fichier de données

En utilisant la fonction `size_t getline(char** linep, size_t* linecapp, FILE* stream)`, écrivez une fonction `void computeExpressions(FILE* input)` prenant en paramètre le flux d'entrée ouvert dans le programme principal et affichant sur le flux standard de sortie, pour chaque chaîne lue, la ligne `Input : chaine_lue`.

Vous veillerez à ce que cette fonction libère totalement la mémoire allouée par la fonction `getline`.

Appelez cette fonction depuis la fonction main et, après compilation, vous devez avoir la sortie suivante pour l'exécution de votre programme :

```
Code$ ./expr_ex1 ../Test/exercice1.txt
Input   : 1 + 2 * 3
Input   : (1+2) * 3
Input   : 1+2^3*4
Input   : (1+2)^3*4
Input   : 1+2^(3*4)
Input   : 1 + 2^3 * 4 * 5
Input   : (1 + 2^(3 * 4)) * 5
```

1.2 Transformation d'une chaîne de caractères en file de *Token*

- Écrivez une fonction `Queue* stringToTokenQueue(const char* expression)` prenant en paramètre une chaîne de caractères et transformant cette chaîne de caractères en file de *Token*. Pour cela, suivez la démarche suivante :
 - Déclarer et construire une variable de type `Queue` qui contiendra le résultat de votre transformation et sera retournée en fin de fonction.
 - Déclarer et initialiser un curseur pour parcourir votre chaîne de caractères : `const char* curpos = expression`
 - Tant que ce curseur ne désigne pas le caractère de fin de chaîne (`'\0'`, correspondant à la valeur 0), rechercher à partir de ce curseur le début et la fin du premier élément syntaxique rencontré. Le début de l'élément syntaxique devra être désigné par `curpos`, la fin de l'élément par le nombre de caractère le composant. Pour cela, il vous faut écrire :
 - Une boucle décalant `curpos` tant que le caractère désigné est un espace ou un retour à la ligne.
 - Une fonction `bool isSymbol(char c)` renvoyant vrai si le caractère `c` est un symbole de l'expression (`'+'`, `'-'`, `'*'`, `'/'`, `'^'`, `'('` ou `')'`).
 - Lorsque `curpos` désigne un caractère qui n'est ni un blanc ni un retour à la ligne, soit c'est un symbole, soit il indique le début d'une valeur numérique. Dans ce cas, il vous faut compter le nombre de caractères composant cette valeur numérique (nombre de caractères jusqu'au prochain symbole).
 - Une fois l'élément syntaxique trouvé, vous pouvez construire le *token* et l'ajouter à la file de *token*. Pour construire le token à partir des informations déterminées ci-dessus, vous devrez utiliser la fonction `Token* create_token_from_string(const char* s, int lg)` ; prenant en paramètre le début du token et le nombre de caractères le composant.
- Modifier votre fonction `void computeExpressions(FILE* input)` afin qu'elle appelle, après avoir affiché la chaîne lue, votre fonction `stringToTokenQueue` et qu'elle affiche le contenu de la file. Pour cela, appelez la fonction fournie dans le fichier main.c `void print_queue(FILE* f, Queue* q)`. Mettez en forme la sortie de votre programme pour obtenir le résultat suivant :

```
Code$ ./expr_ex1 ../Test/exercice1.txt
Input   : 1 + 2 * 3
Infix   : (5) -- 1.000000 + 2.000000 * 3.000000

Input   : (1+2) * 3
```

```

Infix    : (7) -- ( 1.000000 + 2.000000 ) * 3.000000

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

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

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

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

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

```

3. D'une façon générale, lorsque l'on gère dynamiquement des ressources, dans notre cas la mémoire, il est important de pouvoir clairement définir la notion de propriété sur une ressource. Cette notion de propriété permet de savoir quel composant logiciel (module, fonction, ...) est responsable de la libération des ressources. Le rôle d'un module de gestion de collection, dans notre cas le module Queue, étant de stocker une information et de permettre un accès contrôlé à cette information, un tel module n'est pas propriétaire des données stockées. Lors de la destruction de la collection, les ressources dynamiques allouées par le gestionnaire de collection sont libérées par le gestionnaire de collection, les ressources allouées à l'extérieur du gestionnaire de collection (dans notre cas les tokens) ne le sont pas et doivent être libérées par le propriétaire de ces ressources. Afin d'avoir un bilan mémoire nul à la fin de votre fonction `computeExpressions`, programmer le code nécessaire pour libérer l'ensemble des ressources allouées.

2 Algorithme de Shunting-yard

Concepts étudiés :

- Notation infixe et postfixe pour les expressions arithmétiques.
- Utilisation combinée de collections abstraites et génériques.
- Analyse sémantique d'expression arithmétique.

Compétences acquises dans cette partie :

- Savoir convertir une expression infixe en expression postfixe.
- Savoir traduire un algorithme en utilisant des types abstraits de données dont on dispose de la spécification.

L'algorithme de Shunting-yard est une méthode d'analyse et de conversion d'expressions mathématiques de la notation infixe (la notation naturelle pour la plupart des personnes) vers la notation postfixe, aussi appelée [notation Polonaise inversée](#). La notation Polonaise inversée est une notation très bien adaptée à l'évaluation d'expressions arithmétiques puisqu'elle permet de mettre en œuvre des algorithmes d'évaluation efficace vis-à-vis de l'accès mémoire.

Cet algorithme a été inventé par [Edsger Dijkstra](#) en 1961.

À partir de la description de l'algorithme de Shunting-yard accessible à l'url
https://en.wikipedia.org/wiki/Shunting-yard_algorithm
et reproduit ci-dessous :

```
while there are tokens to be read:
    read a token.
    if the token is a number, then push it to the output queue.
    if the token is an operator, then:
        while ((there is an operator at the top of the
            operator stack with
                greater precedence) or (the operator at the
                top of the operator stack has
                equal precedence and
                the operator is left associative)) and
            (the operator at the top of the stack is not a
            left bracket):
            pop operators from the operator stack,
            onto the output queue.
        push the read operator onto the operator stack.
    if the token is a left bracket (i.e. "("), then:
        push it onto the operator stack.
    if the token is a right bracket (i.e. ")"), then:
        while the operator at the top of the operator stack is
            not a left bracket:
            pop operators from the operator stack onto the
            output queue.
        pop the left bracket from the stack.
    /* if the stack runs out without finding a left
       bracket, then there are
       mismatched parentheses. */
if there are no more tokens to read:
    while there are still operator tokens on the stack:
        /* if the operator token on the top of the stack is a
           bracket, then
           there are mismatched parentheses. */
        pop the operator onto the output queue.
```

En utilisant les fonctionnalités offertes par les modules `Token`, `Stack` et `Queue`, écrivez la fonction `Queue* shuntingYard(Queue* infix)` qui transforme l'expression définie par la file de tokens infixe en une file représentant l'expression en notation postfixe.

Vous modifierez votre fonction `computeExpressions` afin qu'elle appelle votre fonction `shuntingYard` et produise l'affichage suivant :

```
Code$ ./expr_ex1 ../Test/exercice1.txt
Input      : 1 + 2 * 3
Infix      : (5) -- 1.000000 + 2.000000 * 3.000000
Postfix    : (5) -- 1.000000 2.000000 3.000000 * +

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

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 * +
```

```

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 *

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 * ^ +

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 * +

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 *

```

3 Évaluation d'expression arithmétique

Concepts étudiés :

- Utilisation du type abstrait de données Stack pour l'évaluation d'expressions.

Compétences acquises dans cette partie :

- Savoir convertir une expression infixe en expression postfixe.
- Savoir traduire un algorithme en utilisant des types abstraits de données dont on dispose de la spécification.

À partir de l'algorithme d'évaluation d'expression en notation Polonaise inversée décrit à l'url https://en.wikipedia.org/wiki/Reverse_Polish_notation#Postfix_evaluation_algorithm et reproduit ci-dessous :

```

for each token in the postfix expression:
    if token is an operator:
        operand_2 <- pop from the stack
        operand_1 <- pop from the stack
        result <- evaluate token with operand_1 and operand_2
        push result back onto the stack
    else if token is an operand:
        push token onto the stack
result <- pop from the stack

```

Écrire la fonction `float evaluateExpression(Queue* postfix)` prenant une expression représentée par une file de tokens en notation postfixe et renvoyant la valeur de l'expression. Pour cela, vous pourrez écrire une fonction `Token* evaluateOperator(Token* arg1, Token* op, Token* arg2)` renvoyant le token résultant de l'application de l'opérateur `op` sur les arguments `arg1` et `arg2`.

Vous modifierez votre fonction `computeExpressions` afin qu'elle appelle votre fonction `evaluateExpression` et produise l'affichage suivant :

```

Code$ ./expr_ex1 ../Test/exercice1.txt
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

```