Compléments d'informatique

Projet 2 : interpréteur d'expressions

6 décembre 2021

Dans ce projet, on vous propose d'implémenter un interpréteur, à la manière du python, pour un langage très simplifié d'expressions mathématiques. Ce langage permet (uniquement) de définir des variables (numériques) sur base d'expressions mathématiques basées sur des variables précédemment définies.

Du point de vue de la programmation, l'objectif est de vous faire utiliser des structures de données, pile, liste et dictionnaire. Vous verrez par la même occasion un algorithme important appelé l'algorithme "shunting-yard" (de la gare de triage) et aurez une petite idée des difficultés liées à l'implémentation des langages de programmation.

Ce projet est à réaliser par **groupe de deux étudiants maximum**. La date limite de remise du projet est indiquée dans Ecampus.

1 Le langage d'expression et l'interpréteur

Le langage, appelons le "Calculator", est un langage simplifié de calcul d'expressions mathématiques. Il n'inclut pas de mécanisme de définition de fonctions et de boucles (et n'est donc pas turing-complet). Il n'existe qu'un seul type d'instruction qui consiste à assigner une expression mathématique à une variable. Une exemple de "code" dans ce langage est le suivant :

```
pi = 3.14159265359
r = 5
aire = pi*r^2
sin(pi/3)^2+cos(pi/3)^2
aire
```

Les expressions mathématiques peuvent s'écrire de la même façon que les expressions mathématiques en C ou en Python sur base des éléments suivantes :

— les parenthèses gauche `'('` et droite `')'` ;
— les opérateurs binaires suivants : `+`, `-`, `*`, `/`, `^` (exposant) ;
— les fonctions mathématiques à un seul argument suivantes : `sin`, `cos`, `exp`, `sqrt`, `ln`, `abs` ;
— des variables (p. ex., `a`, `aire`, `My_var`, `x1`)
— des valeurs numériques entières ou à virgule flotante (p.ex, `1`, `2`, `1.2`, `-.456`)

A la manière du python, le langage est interprété. L'utilisateur peut introduire les définitions de variables les unes après les autres. Chaque nouvel assignation d'une valeur à une variable affiche la valeur de la variable. Il est également possible d'introduire une expression mathématique sans assignation. La valeur de l'expression est alors simplement calculée et affichée, sans être assignée à aucune variable. Cela permet par exemple de consulter directement le contenu d'une variable en introduisant simplement son nom comme expression à évaluer. Le code précédent donnera la session interactive suivante :

```
Calculator 0.0.1
Type "help" for more information
>>> pi = 3.14159265359
3.141593
>>> r = 5
5.000000
>>> aire = pi*r^2
78.539816
>>> sin(pi/3)^2+cos(pi/3)^2
1.000000
>>> aire
78.539816
```

2 Evaluation d'expression

Le processus d'évaluation d'une expression mathématique définie dans une chaîne de caractères est loin d'être triviale. Généralement, on procède en deux étapes :

1. Découpage de la séquence de caractères en une séquence de groupes de caractères appelés "token".
2. Analyse de la séquence de tokens pour calculer la valeur de l'expression.

Lors de la deuxième étape, il est nécessaire de pouvoir accéder aux fonctions ou valeurs associées aux symboles présents dans l'expression (variables, fonctions et opérateurs). Dans le cas des opérateurs, il est également nécessaire de connaître leur précédence et leur associativité (voir plus bas). Cette information est en général stockée dans une table appelée la table des symboles.

2.1 Découpage en tokens

L'analyse lexicale consiste à découper la chaîne de caractères initiale en une séquence de tokens, où un token correspond à un groupe de caractères de la chaîne de départ. Chaque token est défini par trois informations : sa position dans la chaîne, un type, et une valeur associée. Pour notre langage d'expressions, on définit 7 types de token : `SYMBOL`, `OPERATOR`, `EQUAL`, `NUMBER`, `LEFTPAR`, `RIGHTPAR`, et `STOP`. Les tokens de type `EQUAL`, `LEFTPAR`, `RIGHTPAR` correspondent aux caractères `=`, `(` et `)` respectivement et n'ont pas de valeur associée. Le token `SYMBOL` correspond à une chaîne de caractères encodant un nom de variable ou de fonction. Dans notre langage, un symbole est défini comme une lettre suivie d'une séquence de longueur arbitraire de lettres et de chiffres. La valeur associée à un token de type `SYMBOL` est la chaîne de caractère correspondant au nom de la fonction ou de la variable. Le token `OPERATOR` correspond à un opérateur binaire. Dans notre langage, ces opérateurs sont encodés sur un seul caractère (tous les caractères qui ne sont ni des lettres, des chiffres, `.`, `(`, `)`, ou `=` sont des opérateurs potentiels). La valeur d'un opérateur est une chaîne de caractères de longueur unitaire représentant le symbole de l'opérateur. Le token `NUMBER` correspond aux nombres et la valeur associée est la valeur (un double dans l'implémentation) correspondant à ce nombre. Les nombres tels qu'apparaissant dans l'expression peuvent être des nombres entiers, à virgule flottante et être ou non précédés d'un signe `-` pour les nombres négatifs. Enfin, le token `STOP` correspond à la fin de l'expression.

Exemple. Pour l'expression `x-1+sin(2.3)`, l'algorithme d'analyse lexicale produira la séquence de tokens suivante :

```
<SYMBOL,0,"x">, <OPERATOR,1,"-">, <NUMBER,2,1.0>, <OPERATOR,3,"+">,<SYMBOL,4,"SIN">,
<LEFTPAR,7,NULL>,<NUMBER,8,2.3>,<RIGHTPAR,11,NULL>,<STOP,12,NULL>
```

où chaque triplet représente le type, la position et la valeur du token (`NULL` signifiant qu'aucune valeur n'est associée au token).

2.2 Table de symboles

Comme indiqué ci-dessus, la table de symboles est une table qui associe des informations aux symboles (variables, fonctions, et opérateurs) utilisés dans les expressions. Les informations associées à chaque type de symboles sont les suivantes :

- Variables : la valeur courante associée à la variable.
- Fonctions : un pointeur vers la fonction permettant de calculer la valeur de la fonction à partir de son argument. Dans notre langage, on suppose que toutes les fonctions prennent en entrée un double et renvoient un double.
- Opérateurs : un pointeur vers la fonction (prenant deux doubles en entrée et renvoyant un double) permettant de calculer l'opération, ainsi qu'une valeur de précédence et une valeur d'associativité, expliquées ci-dessous.

TABLE 1 – Précédence et associativité des opérateurs

	Précédence	Associativité
`+`	1	gauche
`-`	1	gauche
`*`	2	gauche
`/`	2	gauche
`^`	3	droite

La table des symboles sert aussi à détecter les erreurs dans les expressions : tous les symboles non présents dans la table ne sont pas acceptés dans les expressions et doivent mener à une erreur lors de l'évaluation de l'expression.

Précédence et associativité. La précédence d'un opérateur détermine comment une expression combinant deux opérateurs (sans parenthèses) doit être évaluée. Soient une expression `x op1 y op2 z` et p_1 et p_2 les précédences des opérateurs `op1` et `op2` respectivement. Si $p_1 < p_2$, l'expression doit être évaluée comme `x op1 (y op2 z)` et si $p_1 > p_2$, l'expression doit s'évaluer comme `(x op1 y) op2 z`. Dans le cas où $p_1 = p_2$ (par exemple, dans le cas où `op1`=`op2`), l'évaluation est déterminée selon l'associativité des opérateurs, qui est soit à droite, soit à gauche. Si l'opérateur `op1` est associatif à gauche, l'expression est évaluée en `(x op1 y) op2 z`. Si l'opérateur `op2` est associatif à droite, l'expression est évaluée en `x op1 (y op2 z)`. Pour les opérateurs de notre langage, les valeurs de précédence et d'associativité que nous vous demandons d'utiliser sont précisées dans la table 1.

2.3 Algorithme shunting-yard

L'algorithme shunting-yard [1] prend en entrée une séquence de tokens et calcule la valeur de l'expression correspondante (ou renvoie une erreur dans le cas où l'expression est mal formée). L'algorithme utilise deux piles : une pile d'opérations, notée ci-dessous `Sop` qui contiendra des tokens de type `OPERATOR`, `LEFTPAR`, `RIGHTPAR`, et `SYMBOL` et une pile de valeurs, `Sval` qui contiendra des valeurs de type double. Le pseudo-code de l'algorithme est donné à la figure 1 (en partant des deux piles vides).

L'idée de l'algorithme est de parcourir les tokens itérativement de gauche à droite en modifiant le contenu des deux piles selon les tokens rencontrés. La pile `Sop` retient les opérations (opérateurs binaires ou fonctions) qu'il faudra appliquer ultérieurement, dès que leurs arguments seront calculés, et la pile `Sval` contient les valeurs intermédiaires de sous-expressions déjà calculées. Dans le cas où l'expression est bien formée, la pile `Sop` sera vide

1. `https://en.wikipedia.org/wiki/Shunting-yard_algorithm`

à la fin de l'algorithme, alors que la pile `Sval` contiendra une seule valeur qui sera la valeur de l'expression.

Le pseudo-code suppose que l'expression est correcte. Tous les cas de figure non prévus par cet algorithme correspondent à une expression erronée entrée par l'utilisateur et doivent mener à une erreur. Par exemple, une instruction du type "retirer les valeurs v_1 et v_2 au sommet de `Sval`" suppose que `Sval` contient au moins deux valeurs. Si ce n'est pas le cas, c'est qu'une opérande est manquante dans l'expression pour l'opérateur qu'on devait appliquer sur ces valeurs. Des exemples de traces d'exécution de l'algorithme sont fournis sur Ecampus et seront expliquées au cours.

3 Implémentation

Vous devrez implémenter trois modules (partiellement pour le dernier) :
— Un module `symboltable.c/.h` implémentant la table de symbole.
— Un module `shunting-yard.c/.h` implémentant l'algorithme shunting-yard.
— Un module `calculator.c/.h` implémentant l'interpréteur.

Nous vous fournissons le module suivant, décrit également ci-dessous :
— Un module `tokenizer.c/.h` implémentant pour vous l'analyse lexicale.

Vous sont fournis également des implémentations génériques d'une pile (`stack.c/.h`), d'une liste (`list.c/.h`) et d'un dictionnaire par table de hachage (`dict.c/.h`).

3.1 Fichiers `symboltable.h` et `symboltable.c`

Ce module sert à implémenter la table de symbole. Cette table doit permettre de stocker les valeurs associées aux variables, fonctions, et opérateurs. Un type de données `SymbolTable` opaque doit être défini dans `symboltable.c`. Deux types `FunctionFctType` et `OperatorFctType` pour les pointeurs des fonctions correspondant respectivement aux fonctions et aux opérateurs sont définis dans `symboltable.h` :

```
typedef double (* FunctionFctType)(double);
typedef double (* OperatorFctType)(double, double);
```

qui servent à définir le type des valeurs de retour des fonctions `stGetFunctionFct` et `stGetOperatorFct`.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `symboltable.c` sont les suivantes :

`SymbolTable *stCreate()` : crée une table de symboles vide.

`void stFree(SymbolTable *st)` : libère la mémoire prise par la table `st`.

1. Tant qu'il reste des tokens :
   (a) Lire le token suivant.
   (b) Si le token est de type :
      — `NUMBER` : placer la valeur numérique associée sur `Sval`
      — `LEFTPAR` : placer ce token sur `Sop`
      — `SYMBOL` :
         — si le token suivant est de type `LEFTPAR` : placer le token sur `Sop` (il s'agit d'un appel de fonction)
         — sinon, il s'agit d'une variable : retrouver le symbole dans la table de symboles et placer la valeur associée sur `Sval`.
      — `OPERATOR` : Soit op_1 l'opérateur associé :
         i. Tant qu'il y a un token `OPERATOR` de valeur op_2 au sommet de `Sop` tel que :
            — op_2 a une précédence strictement plus grande que op_1
            — ou op_2 a la même précédence que op_1 et op_1 est associatif à gauche
            retirer op_2 de `Sop`, retirer successivement les valeurs v_1 et v_2 au sommet de `Sval` et placer le résultat de v_2 op_2 v_1 sur `Sval`.
         ii. placer op_1 sur `Sop`.
      — `RIGHTPAR` :
         i. Tant que le token au sommet de `Sop` n'est pas de type `LEFTPAR` :
            — Retirer l'opérateur op au sommet de `Sop`, retirer successivement les valeurs v_1 et v_2 de `Sval` et placer v_2 op v_1 sur `Sval`.
         ii. Retirer le token `LEFTPAR` au sommet de `Sop`
         iii. S'il y a un token `SYMBOL` au sommet de `Sop`, retirer la valeur v au sommet de `Sval` et placer $f(v)$ sur `Sval` où f est la fonction associée à ce symbole dans la table des symboles.
2. Tant qu'il reste un token de type `OPERATOR` de valeur op au sommet de `Sop` :
   — Le retirer de `Sop`, retirer successivement les valeurs v_1 et v_2 de `Sval` et placer v_2 op v_1 sur `Sval`.

FIGURE 1 – Algorithme de shunting-yard pour l'évaluation d'expressions mathématiques

`void stInsertOperator(SymbolTable *st, char *symbol, int prec, int assoc, OperatorFctType opf)`:

introduit un opérateur dans la table. `symbol` est le symbole de l'opérateur, `prec` sa précédence, `assoc` vaut 1 s'il est associatif à droite (0 si à gauche) et `opf` est un pointeur vers une fonction C calculant l'opération.

`void stInsertFunction(SymbolTable *st, char *symbol, FunctionFctType f)` : introduit une fonction dans la table. `symbol` est le nom de cette fonction (dans le langage Calculator) et `f` est un pointeur vers la fonction C à appeler pour calculer la valeur de cette fonction.

`void stInsertVariable(SymbolTable *st, char *symbol, double val)` : insére la variable dont le nom est `symbol` et la valeur `val` dans la table. Si la variable existe déjà, sa valeur doit être mise à jour.

`int stContainsVariable(SymbolTable *st, char *symbol)`
`int stContainsOperator(SymbolTable *st, char *symbol)`
`int stContainsFunction(SymbolTable *st, char *symbol)`
Renvoie 1 si la variable, l'opérateur ou la fonction dénommée par `symbol` se trouve dans la table.

`OperatorFctType stGetOperatorFct(SymbolTable *st, char *symbol)` : renvoie un pointeur vers la fonction associée à l'opérateur dans la table.

`int stGetOperatorPrec(SymbolTable *st, char *symbol)` : renvoie la précédence de l'opérateur (-1 si l'opérateur n'existe pas).

`int stGetOperatorAssoc(SymbolTable *st, char *symbol)` : renvoie l'associativité de l'opérateur (0 si associatif à gauche, 1 si à droite, -1 s'il n'existe pas).

`FunctionFctType stGetFunctionFct(SymbolTable *st, char *symbol)` : renvoie un pointeur vers la fonction associée à la fonction du langage.

`int stGetVariableValue(SymbolTable *st, char *symbol, double *result)` : place dans le `double` pointé par `result` la valeur associée à la variable. Renvoie 1 si la variable existe, 0 sinon.

3.2 Fichiers `shunting-yard.h` et `shunting-yard.c`

Ce module doit implémenter l'algorithme shunting-yard décrit à la figure 1.

Fonction de l'interface. Une seule fonction non statique doit être implémentée dans ce fichier (mais vous pouvez ajouter autant de fonctions auxiliaires statiques que vous le souhaitez) :

`int syEvaluate(Tokenizer *tokenizer, SymbolTable *st, double* solution)` :
Cette fonction doit calculer la valeur de l'expression dont `tokenizer` fournit la séquence de tokens en utilisant la table de symboles `st`, selon l'algorithme shunting-yard décrit ci-dessus. La résultat du calcul doit être stocké dans l'espace pointé par `solution` qui doit être alloué par l'utilisateur. La fonction renvoie 1 si aucune erreur n'est relevée, 0 sinon.

Gestion des erreurs. On souhaite que l'implémentation fournisse la valeur de l'expression dans tous les cas où l'expression est bien formée. En cas d'erreur dans l'expression, au minimum, le programme devrait afficher un message d'erreur et s'arrêter (en faisant `exit(EXIT_FAILURE)`). Les plus motivés pourront écrire cette fonction de manière à ce qu'elle renvoie 0 dans le cas d'une expression erronée, ainsi qu'un message d'erreur le plus explicite possible quant à la nature et la position de l'erreur (voir la section 4). Dans ce cas, la boucle de l'interpréteur telle qu'implémentée dans `calculator.c` rendra la main à l'utilisateur (qui aura alors la possibilité de rentrer une nouvelle expression).

3.3 Fichiers `tokenizer.h` et `tokenizer.c`

Ce module implémente pour vous l'analyse lexicale. Il utilise deux structures de données opaques, `Tokenizer` et `Token`. Les types de token sont définis par l'énumération `TokenType` et correspondent aux valeurs de retour de la fonction `tokenGetType`. Le principe de l'implémentation est qu'après initialisation d'une structure de type `Tokenizer` sur base d'une chaîne de caractères, chaque appel successif à la fonction `tokenizerGetNextToken` sur cette structure renverra le token suivante.

Fonction de l'interface. Les fonctions suivantes sont fournies :

`Tokenizer *tokenizerInit(const char *str)` : initialise une structure de type `Tokenizer` sur base de la chaîne `str`.

`void tokenizerFree(Tokenizer *tokenizer)` : libère la mémoire prise par la structure.

`Token *tokenizerGetNextToken(Tokenizer *tokenizer)` : renvoie le token suivant.

`void tokenizerReset(Tokenizer *tokenizer)` : réinitialise l'analyse lexicale. Le prochain appel à `tokenizerGetNextToken` renverra le premier token de la chaîne.

`int tokenGetPosition(Token *token)` : renvoie la position du début du token dans la

chaîne de caractères.

`TokenType tokenGetType(Token *token)` : renvoie le type du token en argument.

`void *tokenGetValue(Token *token)` : renvoie la valeur du token. Dans le cas des types `T_SYMBOL`, `T_OPERATOR`, le value est de type `char *` et contient le nom de la variable/fonction ou de l'opérateur. Pour le type `T_NUMBER`, la valeur est de type `double *` et pointe vers la valeur double lue. Dans les autres cas, la fonction renvoie `NULL`.

`void tokenFree(Token *token, int freeValue)` : libère la mémoire prise par un token. Si `freeValue` est vrai, la valeur est également libérée. Cette option permet de réutiliser la valeur dans votre implémentation.

`void tokenPrint(Token *token)` : affiche le token. Sert uniquement au débogage.

3.4 Fichiers `calculator.h` et `calculator.c`

Ce module implémente l'interpréteur à proprement parler en intégrant tous les modules précédents. On peut sortir de l'interpréteur en introduisant simplement une instruction vide. Dans ce module, vous ne devez implémenter qu'une seul fonction, `calcInitSymbolTable`. L'autre fonction vous est fournie.

Fonction de l'interface. L'interface ne contient que deux fonctions :

`void calcRepl(void)` : permet de lancer l'interpréteur. REPL signifie "Read-eval-print-loop" et est utilisé dans le domaine des langages de programmation pour nommer la boucle d'un interpréteur qui consiste à lire l'instruction introduite par l'utilisateur, à l'évaluer et afficher le résultat. L'interpréteur lie les instructions sur la sortie standard directement (en ligne de commande). Il est également possible d'exécuter une séquence d'instructions présente dans un fichier sur base du `main.c` fourni en exécutant la commande :

```
./calc < fichiercode.txt
```

`SymbolTable *calcInitSymbolTable()` : Cette fonction initialise la table de symboles utilisée par l'interpréteur. Vous devez compléter cette fonction pour que soient disponibles dans le langage les opérateurs et fonctions précisées dans la section 1.

4 Conseil d'implémentation

L'implémentation de la table de symboles devra passer par l'utilisation d'une table de hachage qui vous est fournie dans (`dict.h/.c`). Si vous souhaitez stocker dans une table

de hachage un pointeur de fonction comme la valeur associée à une clé, il est à noter qu'il n'est pas possible de stocker directement un pointeur de fonction à un emplacement prévu pour un pointeur `void *`. Vous devez donc passer par une structure intermédiaire dont un champ sera le pointeur de fonction qui devra être stocké. Pour la fonction `stFree`, il est nécessaire de libérer également la place prise par ces structures intermédiaires. La fonction `dictFree` ne permet pas de libérer l'espace prise par les valeurs stockées, uniquement les clés sont libérées. Une manière de procéder est de stocker toutes les valeurs allouées dans une liste attachée à la structure pour les libérer lors de l'appel à `stFree`. Vous pouvez pour cela utiliser l'implémentation de liste fournie dans `list.h/.c`.

L'implémentation de l'algorithme shunting-yard est plus compliquée qu'il n'y paraît. Il s'agit de suivre scrupuleusement la description de la figure 1. La gestion des erreurs peut être assez compliquée, ainsi que la gestion de la mémoire (vous avez la responsabilité de libérer l'espace pris par les tokens générés par le tokenizer). Pour les erreurs, comme indiqué précédemment, vous pouvez vous contenter d'arrêter le programme avec un `exit(EXIT_failure)` en cas d'erreur. Un bonus sera attribué aux étudiants qui permettront au programme de continuer malgré une erreur, en vous arrangeant pour que la fonction `syEvaluate` renvoie 0 dans ce cas (pour autant que le programme fonctionne dans le cas où il n'y a pas d'erreur dans l'expression). Pour ce qui est de la gestion de la mémoire, nous ne serons pas très regardants sur les fuites mémoires lors de la correction.

5 Soumission

Le projet doit être soumis via la plateforme de soumission sous la forme d'une archive au format `zip` contenant les fichiers suivants :

- `symboltable.c`
- `shunting-yard.c`
- `calculator.c`
- Le fichier `rapport.txt` complété (qui précise simplement les étudiants et leurs taux de participation).

Vos fichiers seront compilés et testés sur la plateforme de soumission en utilisant le fichier `Makefile` fourni via la commande `make` ou de manière équivalente en utilisant la commande suivante :

```
gcc -o calc main.c tokenizer.c calculator.c symboltable.c shunting-yard.c dict.c stack.c list.c --std=c99 -lm
```

En outre, nous utiliserons les flags de compilation habituels (`-pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter négativement la cote.

Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, l'étudiant (ou le groupe) se verra affecter une cote nulle à l'ensemble des projets.

Bon travail !