Projet 2 : interpréteur d'expressions

Objectif : implémenter un interpréteur d'expressions mathématiques.

```
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
```

Evaluer une expression

Difficulté principale : évaluation d'une expression

$$\texttt{"(3-4)\^{}3+sin(4/5)"} \Rightarrow \text{-0.282644}$$

Deux étapes :

- Découper le texte en "tokens" (sous-chaînes de caractères ayant un sens ensemble)
- Analyser la séquence de tokens pour évaluer l'expression

Découpage en tokens (tokenizer.c)

```
x - 1 + sin(2.3)
```

⇓

```
x|-|1|+|sin|(|2.3|)|
```

⇓

```
<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>
```

7 types : SYMBOL (`x`), OPERATOR (+), EQUAL (=), NUMBER (`2.3`), LEFTPAR (`(`), RIGHTPAR (`)`), STOP (`\n`).

Table des symboles (`symboltable.c`)

Stocke les informations associées aux symboles reconnus dans le langage :

- Variables (`x`,`y1`...) $\Rightarrow$ `double`
- Fonctions (`sin`, `cos`...) $\Rightarrow$ `double (*f)(double)`
- Opérateurs (`+`, `-`...) $\Rightarrow$ `double (*f)(double, double)`
  - $+$ précédence et associativité

Les symboles non présents dans la table ne sont pas permis dans le langage.

Précédence et associativité des opérateurs

`x op1 y op2 z` évalué comme :

- `(x op1 y) op2 z` si précédence de op1 $>$ précédence de op2
- `x op1 (y op2 z)` si précédence de op1 $<$ précédence de op2

`x op y op z` évalué comme :

- `(x op y) op z` si op est associatif à gauche (ex : `-`, `/`)
- `x op (y op z)` si op est associatif à droite (ex : `^`)

Algorithme shunting-yard

Permet de transformer une séquence de tokens en la valeur correspondante.

Basée sur deux piles :

- `Sop` : une pile d'opérateurs (tokens)
- `Sval` : une pile de valeurs (double)

`Sop` contient les opérateurs dont l'évaluation est retardée en attente de l'évaluation de leurs arguments.

`Sval` contient les valeurs intermédiaires déjà calculées.

Voir l'énoncé pour l'algorithme précis.

Algorithme shunting-yard : illustration

Expression : "pi*r^2"

Première phase : on traite les tokens

```
Token: <SYMBOL,0,pi>
  variable => push on Sval
    Sop : |
    Sval: |3.14
Token: <OPERATOR,2,*>
  operator => check Sop
  Sop empty => push token on Sop
    Sop : |*
    Sval: |3.14
Token: <SYMBOL,3,r>
  variable => push on Sval
    Sop : |*
    Sval: |3.14,5.00
Token: <OPERATOR,4,^>
  operator => check Sop
  Token on Sop: *
  => stop and push token on Sop
    Sop : |*,^
    Sval: |3.14,5.00
Token: <NUMBER,5,2.000000>
  number => push on Sval
    Sop : |*,^
    Sval: |3.14,5.00,2.00
Token: <STOP,7,NULL>
```

Deuxième phase : on vide Sop

```
    Sop : |*,^
    Sval: |3.14,5.00,2.00
Token on Sop: ^
  => evaluate
    Sop : |*
    Sval: |3.14,25.00
Token on Sop: *
  => evaluate
    Sop : |
    Sval: |78.54
```

Algorithme shunting-yard : illustration

Expression : `"sin(1*2+3)"`

```
Token: <SYMBOL,0,sin>
  function => push on Sop
    Sop : |sin
    Sval: |
Token: <LEFTPAR,3,NULL>
  leftpar => push on Sop
    Sop : |sin,(
    Sval: |
Token: <NUMBER,4,1.000000>
  number => push on Sval
    Sop : |sin,(
    Sval: |1.00
Token: <OPERATOR,5,*>
  operator => check Sop
  Token on Sop: (
  => stop and push token on Sop
    Sop : |sin,(,*
    Sval: |1.00
Token: <NUMBER,6,2.000000>
  number => push on Sval
    Sop : |sin,(,*
    Sval: |1.00,2.00
Token: <OPERATOR,7,+>
  operator => check Sop
  Token on Sop: *
   precedence => evaluate
    Sop : |sin,(
    Sval: |2.00
  Token on Sop: (
  => stop and push token on Sop
    Sop : |sin,(,+
    Sval: |2.00
Token: <NUMBER,8,3.000000>
  number => push on Sval
    Sop : |sin,(,+
    Sval: |2.00,3.00
Token: <RIGHTPAR,9,NULL>
  rightpar => check Sop
  Token on Sop: +
   operator => evaluate
    Sop : |sin,(
    Sval: |5.00
  Token on Sop: (
   leftpar => remove
    Sop : |sin
    Sval: |5.00
  Token on Sop: sin
  function => evaluate
    Sop : |
    Sval: |-0.96
    Token: <STOP,11,NULL>
```

Fichiers

On fournit :

- `tokenizer.c` : l'analyseur lexicale.
- `dict.c`, `stack.c`, `list.c` : des structures standards de dictionnaire, pile et liste.
- `main.c` : un fichier qui lance l'interpréteur.

Vous devez écrire :

- `symboltable.c` : gestion de la table des symboles
- `shunting-yard.c` : algorithme de shunting-yard
- `calculator.c` : la boucle REPL est fournie. Il faut juste compléter la fonction d'initialisation de la table des symboles.

Difficultés/conseils

`symboltable.c` :

- Utilisez une ou plusieurs tables de hachage
- Attention à la gestion de la mémoire (maintenir une liste des structures créées)
- Voir les slides suivants pour la gestion des pointeurs de fonction.

`shunting-yard.c` :

- Implémentation litérale de l'algorithme de l'énoncé (deux boucles while principales, la première assez longue)
- Le tokenizer fournit les tokens les uns après les autres (`tokenizerGetNextToken`), pas de retour en arrière possible
- Gestions des erreurs (dans les expressions) et de la mémoire assez compliquées. Pas des critères importants pour la note finale.

Remarques sur pointeurs de fonctions (1/2)

Pour donner une valeur de retour de type pointeur de fonction :

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

double myfunction(double x) {
  ...
}

FunctionFctType myfunction2() {
  ...
  return myfunction;
}

void myfunction3() {
  FunctionFctType f = myfunction2();
  double x = f(1.2);
}
```

Remarques sur pointeurs de fonctions (2/2)

En principe, on ne peut pas stocker un pointeur de fonction dans une variable de type `void *` (même si ça peut marcher selon le compilateur).

Et donc, par exemple, ce code est invalide :

```
double myfunction(double x) {
  ...
}
Dict *d = dictCreate(1000);
dictInsert(d, "key", myfunction);
```

L'idée est de passer par une structure intermédiaire :

```
typedef struct MyStruct_t {
  double (* f)(double);
} MyStruct;

MyStruct *mystruct = malloc(sizeof(MyStruct));
mystruct->f = myfunction;
dictInsert(d, "key", mystruct);
```

Modalités

- Par groupes de deux étudiants maximum. Contribution de chacun à préciser dans le rapport.
- Deadline le vendredi 17/12/2021 à 23h59.