Sujet de TD n°10

séance n°17

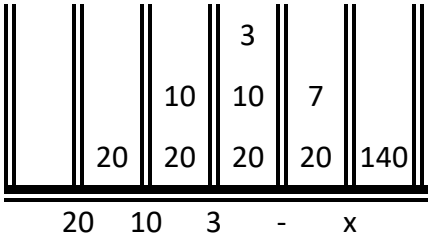
Objectif

L'objectif de cette séance de TD est de préparer le travail d'implantation qui aura lieu au cours des séances de TP n°22, 23 et 24.

Programmation d'un interpréteur NPI

La firme Hewlett-Packard s'est imposée dans les années 1970 auprès de la communauté scientifique et étudiante avec l'introduction de sa calculatrice scientifique HP-35. Elle sortira d'ailleurs en 1988 la HP-42s encore considérée aujourd'hui par les passionnés comme la meilleure calculatrice HP jamais produite. La particularité de ces calculatrices est qu'elles utilisent la notation polonaise inversée (NPI). Celle-ci, appelée aussi notation postfixée car l'opérateur est noté derrière les opérandes, permet d'écrire les formules arithmétiques sans utiliser les parenthèses et ce sans ambiguïté. Par exemple, l'expression "`20×(10−3)`" s'écrit "`20 10 3 − ×`".

L'évaluation de ces expressions est réalisée sur les calculatrices à l'aide d'une pile. Le principe consiste à empiler les opérandes (les nombres) et à chaque fois que l'on rencontre une opération on dépile les opérandes nécessaires (2 pour un opérateur tel que l'addition) et on empile le résultat. Le schéma sur la droite illustre ce principe pour l'expression "`20 10 3 − ×`".



Durant ce TD nous allons nous intéresser à l'implantation d'un interpréteur NPI. Nous nous limiterons ici aux opérations courantes que sont l'addition (+), la soustraction (-), la multiplication (×) et la division (/).

De leur côté, les opérandes, seront définis par l'ensemble des nombres représentables en C++ par le type `double`. Ainsi, 0.5, -1, -2.3, 4, etc. sont tous valables.

L'expression à évaluer sera quant à elle donnée sous la forme d'une chaine de caractères dans laquelle tous les éléments seront séparés par un espace. Par exemple l'expression "`4.2 5+−6−`" sera écrite "`4.2␣5␣+␣−6␣−`".

Algorithme Shunting-yard (gare de triage)

source : cf. https://fr.wikipedia.org/wiki/Algorithme_Shunting-yard

L'algorithme de la gare de triage est un algorithme qui permet de passer d'une notation infixée tel que l'on en a l'habitude en mathématique ("`20×(10−3)`") à une expression postfixée (NPI). Cet algorithme permet ainsi de conserver le principe de l'interpréteur NPI précèdent pour évaluer des expressions infixées mais en passant par une étape de conversion.

Dans les grandes lignes, cette conversion se fait en utilisant en entrée l'expression infixe et pour chaque élément rencontré (un opérande, un opérateur, une parenthèse), on va décider de soit le rediriger directement vers une chaine de sortie, soit mettre l'élément en attente dans une pile (notre gare de triage), soit consommer l'élément sur le dessus de la pile. Le choix de l'action se fait en fonction du type de l'élément et de sa priorité (priorité des opérations).

Une version simplifiée de cet algorithme, adapté à notre besoin, est donnée ci-dessous :

```
Tant qu'il y a des tokens à lire :
  lire le token.
  - si c'est un nombre l'ajouter à la sortie.
  - si c'est un opérateur o1 alors
    1) tant qu'il y a un opérateur o2 sur le haut de la pile et si la priorité
       de o1 est inférieure ou égale à celle d'o2, retirer o2 de la pile pour
       le mettre dans la sortie
    2) mettre o1 sur la pile

  - si le token est une parenthèse gauche, le mettre sur la pile.
  - si le token est une parenthèse droite, alors dépiler les opérateurs et les
    mettre dans la sortie jusqu'à la parenthèse gauche qui elle aussi sera
    dépilée, mais pas mise dans la sortie. Si toute la pile est dépilée sans
    trouver de parenthèse gauche c'est qu'il y a un mauvais parenthésage.

Après la lecture du dernier token, s'il reste des éléments dans la pile il faut
tous les dépiler pour les mettre dans la sortie (il ne doit y avoir que des
opérateurs. Si on trouve une parenthèse gauche alors il y a eu un mauvais
parenthésage).
```

C'est cet algorithme que nous allons utiliser pour permettre à l'interpréteur NPI de travailler sur des expressions données sous une forme infixée.

Travail à réaliser

Au cours des trois séances de TP qui viennent, vous devez implanter une bibliothèque sous la forme d'un couple de fichiers d'entêtes (hpp) et de définitions (cpp). Cette bibliothèque doit contenir l'ensemble des sous-programmes nécessaires au bon fonctionnement du programme principal qui vous est fourni et qui se trouve dans "`calc.cpp`". Ce dernier implante un outil, utilisable en ligne de commande, d'évaluation d'expressions mathématiques postfixées (npi) et infixées. Son usage est décrit ci-dessous.

```
 ./calc --help
This program is a command line tool used to evaluate postfixed and infixed
mathematical expressions
Usage:
  calc [OPTION...]

  -f, --format arg      Specify expression input format [auto | infix |
                        postfix] (default: postfix)
  -e, --expression arg  Input expression
  -d, --debug           Show trace only
  -h, --help            Print usage
```

Exemple d'appels :

```
./calc -f postfix -e "4 5 + 6.3 * 5.2 0.1 / -"
./calc -f infix -e "( 4 - 5 - 7 ) * ( 0.5 - -1 ) / 2.5"
./calc -f auto -e "( 4 + -5 / 3 ) * 1.5" -d
```

En plus de ce programme, deux autres programmes vous sont également mis à disposition. Le premier "`run-test-postfix`" vous permettra de tester votre bibliothèque avec des

expressions postfixées. Le deuxième "`run-test-infix`" vous permettra de la tester avec des expressions infixées. Dans les deux cas, les sorties attendues de ces programmes sont visibles sous la forme de commentaires dans le code. Vous pourrez également les visualiser en lançant directement les exécutables qui vous seront fournis sur AMETICE et en comparant le résultat avec ceux que vous aurez compilé vous-même.

Pour terminer, vous aurez aussi à votre disposition un fichier `Makefile` pour vous permettre de compiler l'ensemble des fichiers. Un complément de cours sur la création et l'utilisation de `Makefile` se trouve sur AMETICE dans la ressource R1.01, section « Compléments de cours ».

Travail bonus

Une fois le travail précédent réalisé, vous pouvez si vous le souhaitez, implanter la vérification du parenthésage pour les expressions infixées. Dans le cas contraire, et pour que votre code compile sans cela, pensez à déclarer la fonction nécessaire dans votre bibliothèque et à la définir. Pour cela vous pouvez l'implanter comme étant un simple "`return true;`". Sans cela, vous ne pourrez pas compiler "`calc.cpp`" !

Listing calc.cpp

```
/**************************************************************************/
// This program is a command line tool used to evaluate postfixed
// and infixed mathematical expressions
// Usage:
//  calc [OPTION...]
//
//  -f, --format arg  Specify expression input format [auto | infix | postfix]
//                    (default: postfix)
//  -e, --expression arg   Input expression
//  -d, --debug       Show trace only
//  -h, --help        Print usage
//
/**************************************************************************/
#include <iostream>
#include <string>
#include "rpn.hpp"
#include "cxxopts.hpp"

bool isAuto(const std::string& format) { return format == "auto"; }
bool isInfix(const std::string& format) { return format == "infix"; }
bool isPostfix(const std::string& format) { return format == "postfix"; }

int main(int argc, char const *argv[]) {
  // Setting up commands lines options
  cxxopts::Options options("calc", "This program is a command line tool used to
evaluate postfixed and infixed mathematical expressions");
  options.add_options()
      ("f,format", "Specify expression input format [auto | infix | postfix]",
cxxopts::value<std::string>()->default_value("postfix"))
      ("e,expression", "Input expression", cxxopts::value<std::string>())
      ("d,debug", "Show trace only", cxxopts::value<bool>()->default_value("false"))
      ("h,help", "Print usage")
  ;
```

```
  // Parsing commands lines
  auto result = options.parse(argc, argv);
  if (result.count("help")) {
    std::cout << options.help() << std::endl;
    exit(0);
  }

  // Asking for input if no expression was given
  std::string input;
  if (result.count("expression")) {
    input = result["expression"].as<std::string>();
  } else {
    std::cout<<"Enter an expression> ";
    std::getline(std::cin, input);
  }

  // Parsing input and building the corresponding expression
  rpn::Expression input_expression = rpn::stringToExpression(input);

  // Define expression format (infix, postfix or auto)
  std::string format = result["format"].as<std::string>();
  if (isAuto(format))
    format = rpn::isValidOperator(input_expression.back())?"postfix":"infix";

  if (!isInfix(format) && !isPostfix(format)) {
    std::cerr<<"/!\\ Error - invalid format specifier ["<<format<<"]"<<std::endl;
    exit(0);
  }

  // If it's an infix expression we convert first
  rpn::Expression expression =
isInfix(format)?rpn::infixToPostfix(input_expression):input_expression;

  // If 'debug' option isn't used, we evaluate the expression
  bool isDebug = result["debug"].as<bool>();
  if (!isDebug) {
    double value = rpn::eval(expression);
    std::cout<<rpn::toString(expression)<<" => "<<value<<std::endl;
  } else {
    std::cout<<"Debug - info"<<std::endl;
    std::cout<<"============"<<std::endl;
    std::cout<<"input: "<<input<<std::endl;
    std::cout<<"format: "<<format<<std::endl;
    std::cout<<"expression: "<<rpn::toString(expression)<<std::endl;
    if (isInfix(format))
      std::cout<<"parenthesis:
"<<(rpn::isWellParenthesized(input_expression)?"ok":"nope")<<std::endl;
  }

  return 0;
}
```