DAEMONLAB

PARSING

- DaemonLab -

Bienvenue.

Vous allez apprendre aujourd'hui une méthode de parsing ***définitive****.*

Après cette journée, parser devrait vous paraître ***beaucoup*** *plus simple, la qualité de vos parseurs sera grandement améliorée et une large partie des difficultés que cette opération implique sera pour toujours derrière vous.*

01 – Détails administratifs

Certains exercices vous demandent d'utiliser le C **et** le C++. Dans ces cas là, les règles sont les suivantes :

Chaque fonction est demandée en double. Vos interfaces C doivent également être utilisable en C++. Les versions C++ sont donc présente pour une question de confort. Usez de la constante de précompilation **__cplusplus** et de la directive **extern "C".** *Votre fichier en-tête doit être compatible avec les deux langages.* Bien entendu, votre implémentation dans un langage **doit** appeler votre implémentation dans l'autre langage, **il ne doit pas y avoir de doublon de code !** Idéalement, implémentez les deux fonctions dans le même fichier.

Vous compilerez avec g++ ou clang++.

Votre rendu doit respecter **strictement** l'ensemble des règles suivantes :

- Il ne doit contenir **aucun** fichier objet. (*.o)
- Il ne doit contenir **aucun** ichier tampon. (*~, #*#)
- Il ne doit pas contenir votre production finale (programme ou bibliothèque)

Le respect de la table des normes est obligatoire.

Vous devez écrire des tests unitaires et exploiter lcov/gcov pour disposer à chaque étape d'une couverture de code avoisinant les 100 %. Un patron de projet avec couverture de code et programmes de tests basés sur assert vous est fourni en annexe.

02 – read_text

Fichier à rendre : dparser.h *.cpp

Réalisez la fonction **dlab_read_text**. Si à str[*index], elle trouve la chaîne de caractère token, alors elle incrémente index de la taille de **token**, et renvoie **true**. À l'inverse, elle n'avance pas index, et renvoie **false**. Les caractères d'espacements sont sautés si la variable globale **gl_dparser_skip_space** est vraie et si ils se trouvent avant le token. Son prototype est le suivant :

```
bool gl_dparser_skip_space = true;

// C
bool dlab_read_text(const char          *str,
                    ssize_t             *index,
                    const char          *token);

// C++ - dlab est un namespace
bool dlab::read_text(const std::string  &str,
                     ssize_t            &index,
                     const std::string  &token);
```

```
char *str = "bouchon de liege";
ssize_t i = 0;

gl_dparser_skip_space = true;
assert(dlab_read_text(str, &i, "bou"));
assert(i == 3);

assert(dlab_read_text(str, &i, "chon"));
assert(i == 7);

assert(dlab_read_text(str, &i, "de"));
assert(i == 10);

assert(!dlab_read_text(str, &i, "lalilou"));
assert(i == 10);
```

03 – read_chars

Fichier à rendre : dparser.h *.cpp

Écrivez la fonction **dlab_read_chars**, qui fait avancer ***index** de façon à traverser tous les caractères **str[*index]** contenus dans **token**. Sa politique à propos des espaces est la même que pour **read_text**. Son prototype est exactement le même que pour la fonction précédente, son nom mis à part.Son prototype est le suivant :

```
// C
bool dlab_read_chars(const char             *str,
                     ssize_t                *index,
                     const char             *token);

// C++ - dlab est un namespace
bool dlab::read_chars(const std::string     &str,
                      ssize_t               &index,
                      const std::string     &token);
```

```
char *str = "Louise Michel";
ssize_t i = 0;

assert(!dlab_read_chars(str, &i, "abc"));
assert(i == 0);

assert(dlab_read_chars(str, &i, "o4uà)_L"));
assert(i == 3);

assert(!dlab_read_chars(str, &i, "seMchl"));
assert(i == 3);

assert(dlab_read_chars(str, &i, "seMichl"));
assert(i == 13);
```

04 – read_symbol

Fichier à rendre : dparser.h *.cpp

En utilisant **read_chars**, implémentez une fonction **read_symbol** dont la capacité consiste à parser un symbole type « C ». C'est à dire un symbole répondant à la syntaxe suivante :

$$[a\text{-}zA\text{-}Z\_]+[a\text{-}zA\text{-}Z\_0\text{-}9]^*$$

Signifiant une suite de caractères alphabétique en minuscule ou majuscule ou underscore en quantité positive suivi d'une suite de caractère alphanumérique ou underscore en quantité positive ou nulle.

La politique de **read_symbol** concernant les caractères d'espacement est la même que celle de read_chars. Si vous avez bien fait votre travail, read_chars devrait s'en occuper presque tout seul. *Faites attention : il ne peut pas y avoir d'espace entre le premier set de caractère et le second, prenez garde à la valeur de* ***gl_dparser_skip_space*** *!*

Le prototype de **read_symbol** :

```
// C
int dlab_read_symbol(const char          *str,
                     ssize_t             *index,
                     char                *output,
                     size_t              outlen);

// C++ - dlab est un namespace
int dlab::read_symbol(const std::string  &str,
                      ssize_t            &index,
                      std::string        &output);
```

Les paramètres **output** servent à contenir les caractères du symbole lu (Attention : pas les éventuels espaces situés avant le symbole…), ils sont écrit par la fonction. Le terminateur \0 est également écrit.

Dans la version C : **outlen** indique la taille disponible dans l'espace mémoire passé en paramètre via la variable **output**. Si **NULL** est passé à **output**, le symbole n'est pas stocké mais *index est tout bien avancé. Si **outlen** n'est pas assez grand et que **output** n'est pas **NULL**, index n'est pas avancé mais la longueur du symbole est tout de même retournée.

La fonction renvoi le nombre d'octets écrits, en incluant le terminateur, 0 si aucun symbole n'est trouvé.

05 – read_cstring

Fichier à rendre : dparser.h *.cpp

Vous allez implémenter une fonction capable de lire une C-string littérale, **read_cstring**, c'est à dire capable d'interpréter les valeurs comme \n. Pour cela, vous allez d'abord créer une fonction capable de lire un unique caractère **read_cchar**.

```
// C
int dlab_read_cchar(const char          *str,
                    ssize_t             *index,
                    char                *output,
                    size_t              outlen,
                    bool                quote);

// C++ - dlab est un namespace
int dlab::read_cchar(const std::string  &str,
                     ssize_t            &index,
                     std::string        &output);

// C
int dlab_read_cstring(const char        *str,
                      ssize_t           *index,
                      char              *output,
                      size_t            outlen,
                      bool              quote,
                      bool              quote = true);

// C++ - dlab est un namespace
int dlab::read_cstring(const std::string &str,
                       ssize_t           &index,
                       std::string       &output,
                       bool              quote = true);
```

La fonction **read_cchar** doit gérer les symboles suivants et valeurs arbitraires :

\a, \b, \t, \v, \n, \v, \f, \r, \e, \\, \', \\", \?, \0n, \xn, \un, \Un

https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences

Et les caractères **UTFf8** :

https://en.wikipedia.org/wiki/UTF-8#Encoding

Le paramètre **quote** indique si la fonction doit consommer ou non une ouverture et fermeture de données pour le caractère ou pour la chaîne de caractère. Si la fermeture est manquante, la fonction renverra -1 pour signaler une **erreur de syntaxe**.

Toutes ces fonctions, si **quote** et **gl_dparser_skip_space** sont vrais, peuvent consommer **avant** l'ouverture des données les caractères d'espacements.

La valeur de retour et les paramètres **output** fonctionnent comme avec **read_symbol**.

06 – read_number

Fichier à rendre : dparser.h *.cpp

Vous allez maintenant implémenter une fonction capable de lire un nombre sous diverses normes. Vous pouvez et devez utiliser **strtol** et **strtod**. Les formats de valeur entières à gérer sont préfixés afin de vous aider à établir leur syntaxe : 0b pour le **binaire**, 0x pour **l'hexadecimal**, 0 pour **l'octal**.

En cas d'absence de préfixe, cela signifie que la valeur sera écrite en **décimal**. Si un point '.' ou un 'e' suit la valeur, cela signifie que celle-ci est un nombre **flottant**, sinon c'est un entier.

```
// C
int dlab_read_number(const char          *str,
                     ssize_t             *index,
                     int                 *integer,
                     double              *real);

// C++ - dlab est un namespace
int dlab::read_number(const std::string  &str,
                      ssize_t            &index,
                      int                &integer,
                      double             &real);
```

La fonction renvoi 0 si elle n'a rien trouvée, 1 si c'est un entier, 2 si c'est un flottant. La politique concernant les espaces est la même que celle des autres fonctions type **read_chars**.

07 – read_value

Fichier à rendre : dparser.h *.cpp

Vous allez maintenant implémenter une fonction capable de lire une donnée littérale, quelque soit sa nature. Cette fonction doit pouvoir lire un entier de n'importe quel format, un flottant de n'importe quel format, un caractère disposant de ses apostrophes de chaque coté ou une chaîne de caractère disposant de ses guillemets.

Le caractère seul sera enregistré comme une chaîne de caractère. La fonction est responsable de l'allocation du champ value de t_data_string si c'est le type qui est trouvé. L'allocation mémoire sera réalisée à l'aide de la fonction **malloc,** en C, comme en C++.

```
typedef enum      e_data_type
{
      INTEGER,
      REAL,
      STRING
}                 t_data_type;

typedef union     u_data
{
      t_data_type type;
      t_data_int  integer;
      t_data_real real;
      t_data_string string;
}                 t_data;
```

```
typedef struct    s_data_int
{
      t_data_type type;
      int         value;
}                 t_data_int;

typedef struct    s_data_real
{
      t_data_type type;
      double      value;
}                 t_data_real;

typedef union     s_data_string
{
      t_data_type type;
      char        *value;
}                 t_data_string;
```

```
// C
bool dlab_read_value(const char      *str,
                     ssize_t         *index,
                     t_data          *output);

// C++ - dlab est un namespace
bool dlab::read_value(const std::string &str,
                      ssize_t           &index,
                      t_data            &output);
```

La fonction renverra vrai ou faux en fonction de la trouvaille d'un élément à lire.

Bien évidemment, les fonctions réalisées précédemment vous seront très utiles.

08 – split

Nous allons aborder brièvement la grammaire avec comme exemple la fonction **split.** Ci-dessous, une grammaire au format **yacc** – un compilateur de compilateur - montrant la grammaire d'une opération d'éclatement d'une chaîne de caractère en éléments séparés par des virgules.

ReadSplit :	SYMBOL ',' ReadSplit	Le symbole ':' indique le début de règle.
|	SYMBOL	Le symbole '|' est un OU logique en Yacc.
;		Le symbole ';' termine la règle ReadSplit.

Notez l'absence de considération pour les caractères d'espacements: ils ne sont pas considérés, ils ne forment qu'un séparateur permettant de mettre fin à la lecture d'un élément, comme lorsque vous utilisez vos fonctions.

De plus, observez le fonctionnement récursif de la règle **ReadSplit**. Vous devriez comprendre maintenant pourquoi on parle de « parseur à descente récursive » et d'où provient la réputation des langages fonctionnels dans le parsing – c'est simplement lié à leur fonctionnement récursif naturel.

Voyez le mot **ReadSplit** comme une fonction et le **SYMBOL** étant un type de donnée final – en l'occurrence, une valeur parsable par la fonction **read_symbol.** Une fonction qui se contenterai de traverser un champ comme "abc, def, ghi, jkl" serait pratiquement la traduction en C de la règle de grammaire ci-dessus :

```
// -1 : Erreur, 0 : Pas trouvé, 1 : Trouvé
int         readsplit(const char                *str,
                      ssize_t                   *index)
{
      int   ret;

      if ((ret = dlab_read_symbol(str, index, NULL, 0)) <= 0)
            return (ret);
      if (dlab_read_text(str, index, ","))
            if (readsplit(str, index) <= 0)
                  return (-1);
      return (1);
}
```

Remarquez que ce code contient déjà de la récupération sur erreur : si il n'y a aucun symbole à lire du tout (par exemple, une chaîne vide, ou ne contenant que des espaces), la fonction renverra 0. Si il y a par contre un symbole, la fonction renverra 1. Si il y a un symbole suivi d'une virgule puis plus rien : il y aura une erreur de syntaxe.

Un message d'erreur adapté serait « Expected symbol after ',' »

09 – read_csv

Fichier à rendre : dcsv.h *.cpp

Prenez garde, le fichier en-tête a changé !

Vous allez programmer une fonction **read_csv** qui chargera un fichier CSV défini par les règles suivantes (Ici présenté au format **yacc**) :

```
Value :     VALUE
|           SYMBOL
;
Line :      Value ';' Line
|           Value
;
Lines :     Line '\n' Lines
|           Line
;
CSV :       Lines
;
```

VALUE étant une valeur parsable par la fonction **read_value** et **SYMBOL** par **read_symbol**. Écrivez la fonction **read_csv** ainsi que ses annexes de la même manière que le chaînage d'éléments séparés par des virgules précédents.

```
// C
void *dlab_new_csv(void);
int dlab_read_csv(const char          *str,
                  ssize_t             *index,
                  void                *data);
t_data *dlab_get_csv(void             *data,
                     size_t           x,
                     size_t           y);

// C++ - dlab est un namespace
class CSV {
      ...
      // operator[] ?
      t_data &get(size_t              x,
                  size_t              y);
      int read(const std::string      &str,
               ssize_t              &index);
      ...
};
```

10 – calculate

Fichier à rendre : dparser.h *.cpp

Êtes-vous prêt ? Vous allez maintenant programmer une fonction évaluant une expression mathématique. Vous l'avez déjà fait, ou avez déjà essayé de le faire par le passé. Cette fois-ci, vous avez cependant des outils redoutables et une considération pour la grammaire qui vont faciliter ce travail d'une manière **radicale**.

Ci-dessous, deux indices, la grammaire d'une opération binaire et la grammaire d'une opération unaire. Vous connaissez déjà l'opération binaire, car l'opérateur ',' virgule en est un.

```
BinaryOperation :   Value BinaryOperator BinaryOperation
|                   Value
;
UnaryOperation :    UnaryOperator UnaryOperation
|                   Value
;
```

Remarquez bien que chaque fonction gère **un** opérateur. Il est cependant possible de gérer dans une fonction un niveau de priorité entier. Le choix vous incombe.

Votre évaluateur d'expression mathématique mettra à disposition les opérateurs binaires +, -, *, /, % ainsi que les opérateurs unaires + et –. Pour finir, il acceptera les parenthèses. La priorité des opérateurs sera gérée par l'ordre d'appel de vos fonctions de parsing : plus la fonction est appelée tôt dans le chaînage des fonctions, moins sa priorité est élevée.

La fonction renverra un **t_data*** contenant le résultat de l'expression mathématique. **NULL** en cas d'erreur de syntaxe. (Parenthèse non fermée, absence d'opérande après opérateur, etc.)

```
// C
t_data *dlab_calculate(const char           *str,
                       ssize_t              *index);

// C++ - dlab est un namespace
t_data *dlab::calculate(const std::string   &str,
                        ssize_t             &index);
```

Vous êtes encouragés à allouer statiquement le **t_data** de **calculate** afin d'éviter une accumulation de **malloc** inutiles. De même, vous avez **interdiction** d'éclater la chaîne façon **split/str_to_wordtab**. *Utilisez les fonctions faites précédemment.*

Remarquez que **calculate** prend un **index** : cela signifie que l'expression peut-être quelque part dans **str**, et non forcement au début… Cela signifie aussi que l'expression peut s'arrêter avant \0. \0 est le terminateur de la **string** : l'expression s'achève quand sa grammaire indique que la fin est atteinte !

11 – evaluate

Fichier à rendre : dparser.h *.cpp

L'écriture de la fonction **evaluate** va vous demander de revoir la fonction **calculate**. La fonction **evaluate** va prendre en paramètre supplémentaire une table de données clefs/valeurs représentant des variables et des valeurs.

Votre expression mathématique précédente va donc pouvoir contenir des variables, contrairement à auparavant ou seuls les littéraux étaient permis. Si une variable n'existe pas, une erreur sera émise.

De plus, vous allez ajouter la gestion de l'opérateur # dont la tache est de concaténer deux opérandes en une chaîne de caractère.

Vous allez également ajouter à chaque opérateur la possibilité de fonctionner sur une chaîne de caractère : la fonction **read_number** sera appelée sur eux afin de les convertir en nombre, et si ce n'est pas possible, *la fonction renverra une erreur de syntaxe.*

Pour finir, vous allez implémenter l'opérateur ternaire a ? b : c (Si a, alors b, sinon c) qui permettra d'augmenter encore les capacités de votre évaluateur d'expression.

```
// C
t_data *dlab_evaluate(const char            *str,
                      ssize_t               *index,
                      const char            **env);

// C++ - dlab est un namespace
t_data *dlab::evaluate(const std::string    &str,
                       ssize_t              &index,
                       const std::map<
                           std::string,
                            std::string
                         >                  &env);
```

Vous pouvez maintenant appeler votre fonction **evaluate** depuis votre fonction **calculate**, car **calculate** n'est plus qu'un cas particulier de la fonction **evaluate**.

12 – Interprète

Fichier à rendre : dparser.h *.cpp

Conservez précieusement le code que vous avez fait jusqu'ici afin de vous en servir comme référence, car vous allez désormais entrer dans un autre monde que celui de l'exécution à la volée. Vous allez maintenant **stocker** ce que vous parsez au lieu de l'exécuter.

Pour contenir l'ensemble des éléments qui nous intéresse, vous allez devoir garder une trace des variables dans votre stockage, et non plus seulement récupérer les valeurs. Augmentez donc d'une constante l'énumération **t_data_type** afin d'y ajouter **VARIABLE**. Le champ **string** peut-être mis à contribution pour y stocker le nom de la variable.

Ce que vous allez gérer dorénavant est un langage ayant la forme suivante :

NomDeVariable = Expression

```
enum OperationType
{
      ADD, SUB, MUL, DIV, MOD, CAT, TERNARY, VALUE
};
class Operation
{
      OperationType type;
      // Le resultat de l'opération sur les opérandes
      t_data      result;
      // operands est vide si c'est un littéral
      std::vector<Operation> operands;
};

// C
bool dlab_parse(const char                  *str,
                ssize_t                     *index,
                void                        *env);

// C++ - dlab est un namespace
bool dlab::parse(const std::string          &str,
                 ssize_t                    &index,
                 const std::map<
                   std::string,
                   Operation
                 >                          &env);
```

A ce stade, la variable **env** n'est plus la liste de variables prédéfinies mais la sortie de la fonction **parse**. Techniquement, il devrait être possible de lire plusieurs chaînes de caractères et d'enrichir env donc à partir de plusieurs fichiers.

L'exercice continue sur la page d'après.

La stratégie la plus efficace pour vérifier que votre parsing fonctionne et que votre structure de sortie est conforme à vos attentes est d'écrire un « **pretty printer** », une fonction capable de restituer ce qui a été lu.

```
// C
ssize_t dlab_restore(char                   *out,
                     size_t                 outlen,
                     void                   *env);

// C++ - dlab est un namespace
bool dlab::restore(std::ostream             &os,
                   const std::map<
                     std::string,
                     Operation
                   >                        &env);
```

Pour pouvoir aller plus loin, il manque plusieurs choses : évidemment, la capacité à **exécuter** ce qui a été lu, mais encore faut-il que cette exécution présente un intérêt. Cela manque de paramètre, et d'opérateur d'appel ( )… pour réaliser des fonctions, mais cela va venir…

Si vous êtes parvenu jusqu'ici en ayant réussi tous les exercices,
vous êtes parés pour le projet BABL !