Dépôt BLIH: PYjour00
Répertoire de rendu : racine du dépôt.

Réalisation d'une classe ProducterStream

Il faut réaliser une classe ProducterStream capable d'alimenter le parseur en faisant abstraction du type de flux.

Fichiers:

- *ProducterStream.hpp*
- *ProducterStream.cpp*

Méthodes:

```
std::string ProducterStream::nextString();
```

Retourne le block lu. Si le flux est vide ou non initialisé, retourne une std::exception. La taille du block n'est pas précisée, elle peut être de 1 octet à beaucoup (cela dépends de la source fichier/console/autre). Attention on ne veut pas que vous retourniez l'intégralité d'un fichier dans le cas d'un loadFile mais bien un morceau.

```
bool        ProducterStream::loadFile(char *path);
```

change le flux à lire, et lit à partir du fichier.

```
bool        ProducterStream::loadStdin();
```

change le flux à lire, et lit à partir de Stdin.
Voir l'exemple d'utilisation avec ConsumerParser pour l'utilisation des fonctions load.*

Réalisation d'une classe de base ConsumerParser qui consomme le flux et assemble les règles de la grammaire

Fichiers:

- *ConsumerParser.hpp*
- *ConsumerParser.cpp*

Méthodes:

```
ConsumerParser::ConsumerParser(ProducterStream &);
```

exemple (attention bien lire tout le TP pour bien le comprendre):

```
#include <iostream>
#include «ProducterStream.hpp»
#include «ConsumerParser.hpp»

using namespace std;

/* lecture d'un fichier CSV correspondant à la BNF

      CSV ::=  identifier  [';' identifier]*
      ;


l'implémentation de []* peut aussi être fait sous forme de macro, genre:
Attention: dans le cas où vous avez défini les macros SAVE_CONTEXT et
RESTORE_CONTEXT …

#define ZERO_OR_MANY(X) \
     while (42) \
      { \
            SAVE_CONTEXT \
            if ((X)) continue;\
      }\
      RESTORE_CONTEXT
```

```
*/

bool  readCSV(ConsumerParser& parse)
{
       SAVE_CONTEXT
      if (parse.readIdentifier())  // identifier
      {
            while (true)   // [';' identifier]*
            {
                  SAVE_CONTEXT
                  if (parse.readChar(';') && parse.readIdentifier())
                        continue;
                  else
                        break;
            }
            RESTORE_CONTEXT // annule le dernier SAVE_CONTEXT
            return true;
      }
      RESTORE_CONTEXT
      return false;
}

int main()
{
      ProducterStream   st;
      ConsumerParser    parse(st);

      st.loadFile(«test.txt»);
      if (readCSV(parse))
            cout << « ca marche » << endl;
      st.loadStdin();
      parse.readChar('\n');
}
```

Les fonctions suivantes respectent la convention de parsage du cours. C’est-à-dire elles consomment du flux lorsque l'entrée est valide, et retourne vrai. Sinon elles laissent intact le flux et retournent faux.
Au départ, le flux est vide. C'est ConsumerParser et dans l'implémentation de ses différentes méthodes, qui va appeler ProducterStream pour alimenter son buffer. Quand le flux est vide, il demande 1 et 1 seul buffer à ProducterStream. Ce buffer est d'une taille quelconque (cela dépend de ce que ProducterStream a pu lire), et est retourné sous forme de string. Le ConsumerParser ne lit donc pas l'intégralité du contenu en une fois. Il consomme d'abord (par les différents appel aux fonctions read*) ce buffer avant d'en demandé un autre. Toutefois comme il (Le Consumer) peut avoir à retourner en arrière par le choix d'une autre alternative, il ne libérera finalement les buffers que lorsqu'on le lui précisera par un 'flush' dans la grammaire.
Les différentes fonctions de parsing vont consommer le flux. Une fois le buffer de nouveau vide, on redemande un nouveau buffer.

```
inline bool       ConsumerParser::flush();
```

Libère ses buffers internes car aucune alternative ne remontera au delà de ce point. Retourne vrai quand la libération s'est bien passé, sinon retourne faux.

```
inline bool       ConsumerParser::peekChar(char c);
```

Teste si le caractère en tête de flux vaut la valeur c et retourne vrai sinon retourne faux.

```
inline bool       ConsumerParser::readChar(char c);
```

Consomme le caractère c en tête de flux, avance de 1 caractère et retourne vrai sinon retourne faux. Utilise peekChar. Correspond à " en BNF.

```
inline bool       ConsumerParser::readRange(char begin, char end);
```

Consomme en tête de flux un caractère dont la valeur est comprise entre begin et end et avance de 1

sinon retourne faux. Correspond à 'a'..'z' en BNF.
ex : readRange('a', 'z'); lit un caractère alphabétique minuscule.

```
bool        ConsumerParser::readText(char *text);
```

Consomme en tête de flux un texte donné de la taille strlen(text) sinon retourne faux. Correspond a « » en BNF.
ex : readText(«ls»); lit la commande ls.

```
bool        ConsumerParser::readEOF();
```

Retourne vrai si on est à la fin du flux.

```
bool        ConsumerParser::readUntil(char c);
```

Consomme en tête de flux les caractères tant que le caractère c n'est pas lu, sinon retourne faux. Correspond à → 'a' en BNF.
ex : si le flux vaut « abcdef », readUntil(«d»); lit «abcd».

```
bool        ConsumerParser::readUntilEOF();
```

Consomme tout le reste du flux. Correspond à → EOF en BNF.

```
bool        ConsumerParser::readInteger();
```

lit la règle BNF suivante sinon retourne faux
readInteger ::= ['0'..'9']+ ;
exemples valides : 0000 123456789 1977 42 666

```
bool        ConsumerParser::readIdentifier();
```

lit la règle BNF suivante sinon retourne faux
readIdentifier ::= ['a'..'z'|'A'..'Z'|'_']['0'..'9'|'a'..'z'|'A'..'Z'|'_']* ;
exemples valides : toto _ __ _0 _42 __toto42

```
inline bool       ConsumerParser::beginCapture(std::string tag);
inline bool       ConsumerParser::endCapture(std::string tag,
                                             std::string& out);
```

permet de capturer des portions de texte du flux.

Un appel à beginCapture sauve en interne la position actuelle du flux. Le paramètre 'tag' permet de sauver plusieurs positions en les différenciant via ce paramètre. Deux appels consécutifs à beginCapture avec le même 'tag' ne stockera la position que du dernier appel.

Un appel à endCapture permet de stocker dans la chaîne 'out' la chaîne de caractères comprise entre la position sauver et référencer par 'tag', et la position actuelle.

Les fonctions retournent faux si elles n'ont pas pu effectuer de capture (buffer interne vide, ou capture vide). Elles retournent vraies autrement.

Exemple: soit le code suivant:

```
//
bool        readExpr(ConsumerParser& parse)
{
      SAVE_CONTEXT
      if (parse.readInteger())
      {
            SAVE_CONTEXT
            std::string      num;
            if (parse.readChar('(')
                  && parse.beginCapture(«toto»)
                  && parse.readIdentifier()
                  && parse.endCapture(«toto», num)
                  && parse.readChar(')')
            )
            {
                  cout << « lue: » << num << endl;
                  return true;
            }
      }
      RESTORE_CONTEXT
      return false;
}
```

avec le contenu suivant «42(toto666)» cela affiche « lue: toto666 »

vous pouvez écrire soit sous forme de MACRO soit sous forme de template le mécanisme de répéteur ***[]?, []*, []+*** afin de vous faciliter la tache.

ParserHttp

Faire une classe ParserHttp dérivée de ConsumerParser capable de lire d'un flux Http avec les deux commandes GET et POST comme décrit dans la BNF suivante.
Note : ici #flush marque quand il est possible de faire un flush...

```
Http ::=    Header '\n'
            Body
;

Header ::=  CMD ' ' URI ' ' VERSION '\n'
      [HeaderList #flush ]*
;

CMD ::= [«GET» | «POST»]
;

URI ::= ['/'|'.'|'?'|'%'|'&'|'='|'+'|':'|'-'|'a'..'z'|'A'..'Z'|'0'..'9']+
;

VERSION ::= «http/1.0»
;

HeaderList ::= HeaderName ':' data
;

HeaderName ::= ['-'|'a'..'z'|'A'..'Z']+
;

data ::= → '\n'
;

Body ::=  → EOF //tout jusqu'à EOF
;
```

Fichiers:

- *ParserHttp.hpp*
- *ParserHttp.cpp*

Méthodes:

```
bool        readHttp(std::map<std::string,std::string>&);
```

Peuple la structure de donnée en paramètre à partir du contenu lue par la règle HeaderList.

Remarque d'ordre général : Dans le cas multi-thread, 1 thread réseau lisant le flux, et 1 thread effectuant le parsing puis les traitements. La classe ProducerStream permet simplement de gérer la concurrence et ainsi réaliser un parseur qui, quelque soit sa complexité, sera insensible à la fragmentation des paquets réseaux lus (mutex sur nextString dans le thread Producer).