Module M3105
TP 5 – Interpréteur

- **Le TP5 doit être fait en binôme, sauf cas particulier. Il sera noté. Lisez tout le sujet avant de commencer !**
- **Une démonstration doit être faite lors de la dernière séance de TP, semaine du 6 Novembre (cf consignes à la fin)**
- **Le code du TP5 doit être déposé sur Chamilo avant le 6 novembre 20h (cf consignes à la fin)**
- **Préambule : - Créez un projet Netbeans M3105-TP5**
  - Copiez dans le répertoire de ce projet tous les fichiers fournis dans /users/info/pub/2a/M3105/tp5 que vous ajouterez ensuite au projet Netbeans

Rappel

On vous fournit un projet permettant d'analyser et d'exécuter des programmes rédigés selon la grammaire suivante :

```
<programme>   ::= procedure principale() <seqInst> finproc <FINDEFICHIER>
<seqInst>     ::= <inst> { <inst> }
<inst>        ::= <affectation> ; | <instSi>
<affectation> ::= <variable> = <expression>
<expression>  ::= <facteur> { <opBinaire>  <facteur> }
<facteur>     ::=  <nombre> | <variable> |  - <facteur> | non <facteur> | ( <expression> )
<opBinaire>   ::= + | - | *  | / | == | != | < | <= | > | >= | et | ou
<instSi>      ::= si ( <expression> ) <seqInst> finsi
```

Ce projet est organisé de la façon suivante :

- **ArbreAbstrait** (.h, .cpp) : contient toutes les classes nécessaires pour représenter l'arbre abstrait
- **Exceptions.h** : contient toutes les exceptions levées dans le projet (déclarations et définitions)
- **Interpreteur** (.h, .cpp) : réalise l'analyse syntaxique et la construction de l'arbre abstrait
- **Lecteur** (.h, .cpp) : permet de lire le programme à exécuter symbole par symbole
- **Symbole** (.h, .cpp) : représente les symboles du programme
- **SymboleValue** (.h, .cpp) : représente les symboles valués (les feuilles de l'arbre abstrait : entiers et variables)
- **TableSymboles** (.h, .cpp) : la table des symboles valués
- **main.cpp** : le programme principal qui instancie un interpréteur pour analyser et exécuter un programme
- **motsCles.txt** : fichier texte qui contient les mots clés du langage (un mot par ligne… à compléter si besoin)
- **programme.txt** : un exemple de programme pouvant être exécuté

1. Ajouter des Instruction

Il faut modifier/compléter le langage pour qu'il puisse reconnaître et exécuter les instructions suivantes :

```
<inst> ::= <affectation> ; | <instSiRiche> | <instTantQue> | <instRepeter> ; | <instPour> |
           <instEcrire> ; | <instLire> ;
<instSiRiche> ::= si (<expression>) <seqInst> { sinonsi (<expression>) <seqInst> } [sinon <seqInst>] finsi
<instTantQue> ::= tantque ( <expression> ) <seqInst> fintantque
<instRepeter> ::= repeter <seqInst> jusqua ( <expression> )
<instPour>    ::= pour ( [ <affectation> ] ; <expression> ; [ <affectation> ] ) <seqInst> finpour
<instEcrire>  ::= ecrire ( <expression> | <chaine> { , <expression> | <chaine> } )
<instLire>    ::= lire ( <variable> { , <variable> } )
```

Pour chaque nouvelle instruction ci-dessus, il faudra procéder ainsi :

- Rajouter une méthode correspondant à l'instruction dans la classe **`Interpreteur`**. Exemple : **`Noeud* InstTantQue() ;`**
- **Dans un premier temps**, programmer cette méthode pour qu'elle ne réalise que l'analyse syntaxique… La méthode ne construira pas de nœud et se terminera donc par : **`return nullptr;`**
- Modifier les méthodes **`seqInst()`** et **`inst()`** de l'**`Interpreteur`** pour que la nouvelle instruction soit prise en compte
- Vérifier que l'analyse syntaxique de la nouvelle instruction fonctionne correctement
- Dans **`ArbreAbstrait.h`** et **`ArbreAbstrait.cpp`**, rajouter la déclaration et la définition d'un nœud permettant de représenter l'instruction dans l'arbre (ex : **`class NoeudInstTantQue`**).
  Vous devez bien réfléchir aux points suivants :
  - Quelles informations pertinentes doivent-elles être stockées dans ce nœud ? (qui sont ses fils ?)
  - Comment sera-t-il construit lors de l'analyse syntaxique ?
    - En une fois (*via* le constructeur du nœud) pour les nœuds ayant un nombre constant de fils.
    - Ou « incrémentalement » (besoin d'une méthode ajoute) pour les nœuds ayant un nombre de fils variable.
  - Quelle est la sémantique de l'instruction et donc que faut-il faire dans la méthode **`int executer();`**

  Remarque *: Pour simplifier, on ne se préoccupera pas de programmer les destructeurs des nœuds de l'arbre… Dans la « vraie vie » on utiliserait sans doute des « smart pointers » pour régler ce problème (si ça vous intéresse, allez jeter un coup d'œil à la bibliothèque <memory> de la librairie standard.*
- **Dans un 2ème temps**, modifier la méthode de l'interpréteur pour qu'en plus de l'analyse syntaxique de l'instruction, elle instancie un nœud représentant cette instruction et le renvoie en résultat
- Vérifier maintenant que la nouvelle instruction est correctement exécutée

Remarques :

- Dans l'instruction **ecrire**, on a introduit un nouveau type de symbole, les chaînes de caractères (exemple **"i="**), afin de pouvoir par exemple exécuter une instruction telle que : **ecrire("i = ", i) ;**
- Les chaînes seront considérées comme des symboles valués qu'il faudra donc insérer dans la table des symboles lors de l'analyse syntaxique (comme on le faisait pour les entiers et les variables).
- Dans la méthode exécuter de l'instruction **ecrire**, vous aurez alors besoin de savoir si un **Nœud*** pointe sur une chaîne dans la table des symboles, afin de pouvoir traiter ce symbole différemment (il ne faudra pas écrire sa valeur mais sa chaîne de caractères…)…
  Pour savoir si un pointeur **p** de type **Nœud*** pointe sur une chaîne, vous utiliserez l'opérateur **typeid** de C++ (rajouter **#include <typeinfo>** pour y avoir accès) :
  ```
  // on regarde si l'objet pointé par p est de type SymboleValue et si c'est une chaîne
  if ( (typeid(*p)==typeid(SymboleValue) &&  *((SymboleValue*)p)== "<CHAINE>" ) …
  ```

2. Récupérer après une erreur de Syntaxe

Dans la version initiale qui vous est fournie, l'interpréteur s'arrête dès qu'une exception **SyntaxeException** est levée au cours de l'analyse syntaxique. Cette exception est traitée dans le programme principal pour afficher le message d'erreur associé.
On aimerait que la détection d'une erreur de syntaxe n'arrête pas l'**Interprete** et qu'il soit capable de poursuivre l'analyse après la détection d'une erreur.
Pour cela, il faut traiter l'exception dans les méthodes de **l'Interprete** et essayer, après une exception, de « récupérer » en faisant avancer le lecteur jusqu'à un symbole courant qui permette de poursuivre l'analyse.
Deux options pour faire cela :

- **Option « à gros grain »** (qui ne donnera pas d'excellents résultats mais qui est rapide à faire) : on traite l'exception dans la règle **<inst>** et on fait avancer le symbole courant jusqu'au prochain début d'instruction supposé
- **Option « à grain plus fin »** (qui demande donc plus de travail…) : on traite l'exception dans chaque type d'instruction (chaque règle de la grammaire) et on fait avancer le symbole courant pour tenter de récupérer l'erreur à l'intérieur de la règle en cours et ainsi poursuivre l'analyse de cette règle jusqu'au bout.

Quelle que soit l'option choisie, il faudra faire en sorte que l'arbre abstrait soit vide dès lors qu'une erreur de syntaxe a été détectée afin de ne pas tenter d'exécuter un arbre incomplet ou incorrect produit par une analyse comportant des erreurs de syntaxe.

3. Vous interprétiez ? Et bien compilez maintenant…

On souhaite pouvoir traduire et compiler le programme analysé dans un autre langage que vous choisirez librement (Ada, C/C++, Java, …). Cette opération est en fait assez simple : il suffit de doter chaque type de nœud de l'arbre (y compris **SymboleValue**) d'une « opération » (au sens *pattern composite*) capable d'écrire l'instruction décrite par ce nœud dans le langage cible choisi.
Par exemple, si l'on choisit de produire du C++, voici ce que donnerait cette méthode pour NoeudInstSi :

```
void NoeudInstSi::traduitEnCPP(ostream & cout, unsigned int indentation) const {
  cout << setw(4*indentation) << "" << "if (";      // Ecrit "if (" avec un décalage de 4*indentation espaces
  m_condition->traduitEnCPP(cout, 0);               // Traduit la condition en C++ sans décalage
  cout << ") {"<< endl;                             // Ecrit ") {" et passe à la ligne
  m_sequence->traduitEnCPP(cout, indentation+1);    // Traduit en C++ la séquence avec indentation augmentée
  cout << setw(4*indentation) << "" << "}" << endl; // Ecrit "}" avec l'indentation initiale et passe à la ligne
}
```

La méthode reçoit en paramètre le flux sur lequel on veut écrire la traduction (**std::cout** ou un fichier texte que l'on aura ouvert par ailleurs… exemple : **ofstream cout("traduction.cpp")**).
Elle reçoit également un paramètre indentation qui permet de faire du « pretty print », c'est-à-dire de bien présenter le code produit en augmentant l'indentation lorsque c'est nécessaire (par exemple, dans l'instruction **si**, la séquence d'instruction du **si** est produite avec une indentation augmentée de un afin que les instructions de la séquence apparaissent décalées par rapport au **if**).

Afin que le code ainsi produit puisse vraiment être compilé, il faut que la traduction de l'arbre en C++ soit « insérée » dans une déclaration de programme principal pour le langage cible, déclaration qui devra également contenir la déclaration de toutes les variables du programme (variables dont on trouvera les noms dans la table des symboles !).

Pour cela, vous ajouterez à la classe **Interprete** une méthode qui sera chargée de ce travail d'«habillage»:

```
void Interpreteur::traduitEnCPP(ostream & cout, unsigned int indentation) const {
  cout << setw(4*indentation) << "" << "int main() {" << endl; // Début d'un programme C++
  // Ecrire en C++ la déclaration des variables présentes dans le programme...
  // ... variables dont on retrouvera le nom en parcourant la table des symboles !
  // Par exemple, si le programme contient i,j,k, il faudra écrire : int i; int j; int k; ...
  getArbre()->traduitEnCPP(cout,indentation+1); // lance l'opération traduitEnCPP sur la racine
  cout << setw(4*(indentation+1)) << "" << "return 0;" << endl ;
  cout << setw(4*indentation) << "}" << endl ; // Fin d'un programme C++
}
```

Cette question sera considérée comme réussie si le code produit est agréable à lire et peut être compilé tel quel sans erreur !!!

4. Testez

Pour chaque instruction que **vous** avez ajoutée à notre langage, on vous demande d'écrire un ou plusieurs petits programmes de test indépendants, **commentés**, qui permettront de vérifier facilement que l'instruction fonctionne correctement.
Exemple :

TestSi.txt

```
# Test du si
# Résultat attendu :
# test1 = 1
# test2 = 1
procedure principale()
  test1 = 0 ;
  si ( 1 )
    test1 = 1 ;
  finsi
  test2 = 1 ;
  si ( 0 )
    test2 = 0 ;
  finsi ;
finproc
```

Ce sont ces fichiers que vous utiliserez pour faire la démonstration de votre travail lors de la dernière séance de TP du module.

On vous demande également, pour **une seule instruction de votre choix**, de mettre en place des tests unitaires avec **`CppUnit`** afin d'automatiser le test de cette instruction. Ces tests unitaires devront lancer l'interpréteur sur plusieurs petits programmes (tel que le « TestSi.txt » de l'exemple précédent) et faire des assertions pour :

- vérifier qu'à la fin les variables ont bien la valeur attendue
- vérifier que si un programme de test comportait des erreurs de syntaxe (volontaires !), l'exception **`SyntaxeException`** a bien été levée
- etc…

5. Priorité des Opérateurs

L'analyse syntaxique des expressions, telle qu'elle vous a été fournie, ne permet pas de prendre en compte la priorité des opérateurs. Pour remédier à cela, il faut décomposer la règle <expression> de la façon suivante :

```
<expression> ::= <terme> { + <terme> | - <terme> }
     <terme> ::= <facteur> { * <facteur> | / <facteur> }
   <facteur> ::= <entier> | <variable> | - <expBool> | non <expBool> | ( <expBool> )
   <expBool> ::= <relationET> { ou <relationEt> }
<relationEt> ::= <relation> { et <relation> }
  <relation> ::= <expression> { <opRel> <expression> }
     <opRel> ::= == | != | < | <= | > | >=
```

Vous remarquerez également que cette décomposition fait apparaître la notion d'expression booléenne (règle <expBool>) que l'on pourra utiliser à la place d'expression dans les instructions qui ont besoin de cette notion (si, tantque, répéter, …).

Intégrez ces nouvelles règles dans l'interpréteur en notant bien que pour faire cela, **il n'y a pas besoin de créer de nouvelles classes dans l'arbre abstrait** : toutes ces règles ne produiront, comme la règle **`<expression>`** initiale, que des arbres comportant des nœuds de type **`SymboleValue`** et **`NoeudOperateurBinaire`**.

Ecrivez des programmes de tests pour vérifier que les priorités des opérateurs sont bien respectées après ces modifications.

6. Questions Ouvertes – Enrichir le langage de l'interpréteur

- **Facile – Ajouter de nouvelles instructions**
  - Pré/Post Incrémentation/Décrémentations (`i++, --j, …`)
  - Instruction Selon (équivalent du switch) :

```
selon (i)
  cas 1 : <seqInst>
  cas 2 : <seqInst>
  …
  defaut : <seqInst>
finselon
```

  - … et/ou tout ce que vous pourrez imaginer !

- **Moins Facile – Gérer des variables pouvant contenir des chaînes**

```
i = "hello" ;
i = i + " World" ; # opérateur + de concaténation
ecrire(i) ;
```

- **Plutôt difficile – Pouvoir déclarer et appeler des procédures**

```
procedure p1 (i,j)
  <seqInst>
finproc

procedure p2 (a,b,c)
  <seqInst>
finproc

procedure principale ()
  appel p1(1,2) ;   # mot-clé appel nécessaire pour garder la grammaire LL(1)
  appel p2(3,4,5) ;
finproc
```

7. Rendu du Projet et Démo

- Vous devrez déposer votre projet sur Chamilo (module M3105, travaux) **avant le lundi 6 novembre 20h00**.
  Pour cela, dans Netbeans :
  - Faites d'abord un « clean » de votre projet (onglet projet, clic-droit sur votre projet, « More Build Commands », « Clean Project »
  - Puis faites un export au format zip de votre projet (menu Files / Export Project / To zip…) en donnant à votre archive les noms des membres du binôme (ex : DUPONT_DURAND.zip)
  - Enfin, déposez l'archive obtenue sur Chamilo

- **La séance de démo de votre projet aura lieu la semaine du 6 Novembre avec votre enseignant de TP.**
  - Prévoyez des petits programmes de test simples qui illustrent les fonctionnalités que vous avez implémentées et montrez-les dans l'ordre de grille d'évaluation (à regarder sur Chamilo). Chaque démo dure au plus 15mn, questions comprises.
  - Vos enseignants poseront des questions à chaque membre du binôme… Si l'un des membres a clairement moins travaillé que l'autre sur le projet, les deux membres n'auront évidemment pas la même note à la fin !