# TP 2 : créer et manipuler l'arbre de syntaxe abstraite (AST)

**Description du travail à réaliser** Dans ce TP, vous devrez écrire du code java pour créer l'arbre de syntaxe abstraite de votre langage de programmation; et commencer à écrire des algorithmes qui parcourent ces arbres. En cours, vous avez vu ces manipulations pour l'exemple d'un langage minimal contenant des arbres binaires.

Il s'agit donc de faire la même chose, mais avec les élémenst d'un langage de programmation.

**Dans cette aventure, vous n'êtes pas seul...**

Sur Celene, une archive pour le TP2 vous est fournie, qui contient l'architecture de base du code attendu. Vous y trouverez l'équipement suivant :

- Un *programming language kit*, c'est-à-dire un fichier ANTLR (`plk.g4`) à compléter pour développer votre langage. Il contient le début d'une syntaxe pour les expressions (vous devez donc l'enrichir, et rajouter les instructions).
- Un package "ast" contenant la structure de base des classes java nécessaires à la constructions de l'arbre :
  - Une classe abstraite pour les nœuds de l'arbre, dont devront hériter toutes les constructions.
  - Une classe abstraite pour les expressions, (qui permettra par la suite d'écrire des algorithmes spécifiques aux arbres d'expressions.)
  - Une classe concrète pour les deux types de nœuds déjà présents dans la grammaire fournie (entiers et opérations unaires dans `plk.g4`). Également une énumération pour les opérateurs disponibles.
  - Une classe `AstBuild` permettant de récupérer l'arbre de syntaxe ANTLR pour créer l'AST avec les classes mentionnées plus haut. C'est une classe qui fait la "plomberie" entre le parseur généré et nos classes à nous; elle est donc cruciale car toute extension du langage devra être prise en compte ici (et implémentée ou non dans l'ast selon le type d'extension).

    Cette classe implémente le patron de conception visiteur conçu pour ANTLR.

  - Une interface de visiteur pour les classes de notre AST. Les algorithmes sur nos arbres seront des classes implémentant cette interface. On pourra isoler une interface spécifique pour des visiteurs d'expressions uniquement, si nécessaire.
  - Un exemple de visiteur concret conçu pour transformer l'ast en chaîne de caractères lisible `AstPrinter`; il est à compléter.

- Une classe `Main` où sont implémentées les fonctionnalités de base utilisant les méthodes des librairies ANTLR, de façon à générer à partir de caractères (lus sur l'entrée standard) un arbre de syntaxe (`ParseTree`) qui pourra recevoir notre visiteur `AstBuild` et créer l'objet correspondant à l'ast.

  Est également illustrée l'instanciation de l'AstPrinter pour pouvoir être testée rapidement.

- Un script bash[1] `build.sh` permettant de générer le parseur (les fichiers sont générés dans un package `parser` pour l'occasion) et de compiler les classes java. Pour l'utiliser, il est nécessaire d'avoir mis à jour son `bashrc` comme indiqué dans le TP précédent[2].

- Un script `clean.sh` pour supprimer tous les fichiers générés.

- Remarque : n'hésitez pas à vous approprier ces scripts en les adaptant à vos besoins ou préférences de développement et de tests, en ajustant par exemple le `MakeFile` fourni au TP précédent pour plus de maniabilité.

**Premiers tests**

Dans un terminal Unix, placez vous dans le répertoire du TP2 (une fois extrait de l'archive fournie), et lancez `./build.sh` après vous être assuré d'avoir la configuration adéquate.

Ensuite, lancez la commande `java Main`, qui devrait vous laisser ouverte l'inivite de commande.

Tapez une expression à parser. Les seules expressions prises en charge dans le code fourni sont les entiers littéraux. Terminez votre saisie d'un entier par 'Entrée' puis 'Ctrl+D'. L'entier devrait être recopié, car le programme a appelé le visiteur qui affiche l'ast.

---

[1] Si vous préférez utiliser uniquement IntelliJ : Dans la liste des fichiers (présente sur le panneau latéral), faire un clic droit sur le fichier .g4 de votre grammaire, puis 'Generate Antlr recognizer', (ou Ctrl+Maj+G). Un répertoire `gen` est créé. (Pour votre programme, il pourra être nécessaire de renommer les packages).

  - Il vous faudra importer les librairies Antlr4 pour Java. Pour cela, rendez-vous dans `Files > Project Structure > Libraries`.
  - Puis avec Maven, recherchez une librairie Antlr4 (par exemple :antlr4-4.11), puis installez-la.

[2] Ajouter la ligne `export CLASSPATH=".:/usr/local/lib/antlr-4.13.1-complete.jar:$CLASSPATH"` en adaptant éventuellement la version et le chemin s'ils sont différents.

Dans la suite de vos tests, vous pourrez écrire les textes à parser dans des fichiers `test.plk` et utiliser le programme en redirigeant le fichier vers l'entrée standard en le lançant avec `java Main < test.plk`.

# 1 Première mission

Répétez le test ci-dessus avec une expression unaire au lieu d'un entier. Elles sont en effet déjà définies dans `plk`, avec plusieurs opérateurs possibles. Vous pouvez donc tester des expressions telles que `-2`. Mais l'afficheur ne prend pas encore en charge ce type de nœuds.

1. Modifiez le visiteur `AstPrinter` pour pouvoir afficher les expressions unaires sous formes de chaînes de caractères.
2. Modifiez l'`AstBuild` pour traiter le cas des expressions parenthésées. Si vous n'y parvenez pas ou ne voyez pas du tout comment faire (mais cherchez un tout petit peu quand même, je l'ai montré en cours), regardez le code de la démo du cours.
3. Répétez les tests avec des expressions unaires comme `-(-(-(-12)))`.
4. Intégrez la prise en charge de l'opérateur `not` et `abs`

# 2 Deuxième mission : Extension du langage

## 2.1 Spécification de la gramamire

**Conseil** Dans cette partie du TP, vous devez ajouter des éléments à votre grammaire. Il est vivement recommandé de les ajouter petit à petit (par exemple, les constructions relatives au fonctions peuvent être laissées de côté dans un premier temps). Vous pouvez bien sûr récupérer ce que vous avez fait lors du TP précédent. Travaillez maintenant le fichier `plk.g4` pour ajouter :

**Lexèmes**

- Les constantes booléennes

**Syntaxe**

- Expressions :
  - Constantes booléennes, du coup
  - Variables
  - Expressions binaires

  - Appels de fonction : `f(x,y,...)`

- Instructions :
  - Déclaration de variable
  - Affectation
  - Liste d'instructions
  - Boucle while
  - Conditionnelle
  - Retour de valeur (pour les fonctions)

## 2.2 Création des classes de l'AST

Définissez des classes Java correspondant à votre langage :

- Une sous-classe abstraite de Nœud pour les instructions.
- Une classe concrète pour chaque catégorie syntaxique décrite en section 2 (pour chaque nœud de l'AST générique).

## 2.3 Rassembler le parseur et l'AST

Récupérez le code java de votre parseur (qui supporte l'option Visiteur). Si votre langage s'appelle `Langage`, vous devriez disposer entre autres des fichiers suivants : `plkLexer.java, plkParser.java, plk.tokens, plkVisitor.java, plkBaseVisitor.java`.

### 2.3.1 Identification du code généré par ANTLR

Parcourez le fichier `plkBaseVisitor` et `plkParser`. Remarquez que pour chaque catégorie syntaxique que vous avez inscrite dans votre fichier `.g4`, une classe statique a été créee dans le parseur, et que le visiteur a une méthode de visite pour chacune de ces classes. Ces classes sont `<Règle>Context`, et héritent de `ParserRuleContext`.

Dans les classes statiques du `LangageParser`, vous avez des méthodes vous permettant de récupérer :

- Des nœuds terminaux (pour les entiers ou les variables), que vous pouvez traiter comme des `String` grâce à la méthode `getSymbol()` (cf doc Antlr Java sur Oracle)
- Des `Context`, ce sont les sous arbres, sur lesquels vous pourrez appeler vos visiteurs pour les diffuser (ils se comporteront comme une fonction récursive sur les arbres).
- Des listes de `Context` (quand il y a plusieurs sous-arbres), que vous devrez parcourir pour les traiter comme au point précédent. (Se référer si besoin à la collection `List` de Java.

#### 2.3.2 Codage

Modifiez la classe `AstBuild` pour créer l'AST adapté, en vous servant des points précédents et des exemples déjà codés.

# 3 Algorithmes sur l'AST

## 3.1 Supporter le patron visiteur

Pour que le nouvel AST soit compatible avec les algorithmes implémentés sous forme de visiteurs, il faut au préalable s'assurer des points suivants :

- Dans les nouvelles classes de votre AST, vous devez rajouter les méthodes `accept` impérativement pour que le code compile, car c'est imposé par la classe abstraite `Node`. Ces méthodes sont toutes identiques et servent juste à ce que le visiteur soit appelé correctement sur chacun des objets.
- L'iterface `Visiteur` pour l'AST doit comporter une déclaration de méthode `visit` pour chaque sous classe concrète de l'AST. Si vous en oubliez, des sous-arbres entiers du programme seront oubliés par les algorithmes.

## 3.2 Premier visiteur : afficher le code source

La classe `AstPrinter` ne compilera pas tant que vous n'aurez pas implémenté toutes les méthodes déclarées dans l'interface lors du point précédent.

Complétez ce visiteur de façon à ce qu'il retourne une chaîne représentant l'AST de façon lisible.

Faites en sorte que le code respecte une indentation naturelle des blocs dans les conditionnelles et les boucles.

# 4 Extension du langage — Deuxième partie

Dans cette section, on vous demande de rajouter des constructions au langage, sans rajouter pour autant de classes à l'AST.

Pour les extentions demandées, vous avez le droit de modifier uniquement :

- Le fichier .g4, évidemment, pour permettre les nouvelles formes grammaticales.
- L'`AstBuild` qui devra implémenter ces constructions en utilisant les nœuds déjà définis dans l'AST.

Vous devez rajouter la possibilité pour l'utilisateur d'écrire :

1. Des boucles for (il vous faudra avoir déjà implémentées les boucles while)
2. Des affectations/déclarations de la forme `int x = 1;` (il faut avoir les déclarations, et les affectations)

3. Des incrémentations et décrémentations du type `x++;` (qui seront considérées uniquement comme des instructions, et non comme des expressions, pour plus de simplicité).