LOG3210 - Elément de langages et compilateur
TP1 : Grammaire et analyseur syntaxique

Doriane Olewicki – Chargée de cours
Gérard Akkerhuis – Chargé de laboratoire (Gr. 01 et 02)
Hakim Mektoub – Chargé de laboratoire (Gr. 03)

Hiver 2022

1 Objectifs

- Se familiariser avec JavaCC;
- Utilser un analyseur lexical;
- Construire un analyseur syntaxique descendant.

2 Travail à faire

JavaCC (Java Compiler Compiler) est utilisé afin de générer les analyseurs lexical et syntaxique en Java à partir de règles décrites dans un fichier `.jjt`. Le fichier (`Template.jjt`) est divisé en deux sections : une pour l'analyse lexicale et une pour l'analyse syntaxique. L'analyseur lexical permet de séparer le programme fourni en entrée en jetons ("token"). Ces jetons sont généralement les mots clés, les opérateurs, les identificateurs, les caractères spéciaux, etc., définis dans le langage.

L'analyseur syntaxique permet quand à lui de déterminer la validité du programme donné en entrée. Il analyse les jetons retournés par l'analyseur lexical et vérifie si ces derniers respectent les règles définies dans la grammaire. La grammaire définit la syntaxe du langage – l'analyseur syntaxique permet donc de vérifier que la suite de jetons qui constitue le programme en entrée respecte bien la syntaxe du langage.

La grammaire JavaCC qui vous est fournie décrit un langage permettant d'assigner des valeurs à des variables et d'effectuer des opération arithmétiques élémentaires et d'exécuter certaines fonctions mathématiques.

Modifiez la grammaire JavaCC de façon à ce qu'elles reconnaissent toutes les structures ci-dessous en terminant le corps des fonctions identifiées et en ajoutant au besoin de nouvelles fonctions. Le dossier `test/PrintTest/data` contient des exemples de codes valides et invalides du langage.

2.1 Les boucles while et do-while

Les noms-terminaux `WhileStmt` et `DoWhileStmt` doivent être utilisés respectivement pour les boucles while et les boucles do-while. Les deux structures suivantes doivent être acceptées par votre grammaire.

Listing 1: Boucle While (deux exemples)

```
while (expression) {
    block
}

while (expression)
    statement
```

Listing 2: Boucle Do-While

```
do {
    block
} while (expression);
```

2.2 Assignations

Les assignations se font à des variables SANS MOTS CLÉS DEVANT. Ces variables peuvent recevoir des expressions booléennes, numériques et une fonction anonyme.

Listing 3: Exemples d'assignation

```
variable = a + 1;
var2 = true;
```

pour ce TP, il n'y pas de vérification de types. Donc, oui c'est accepté "a = 1; a = true;" par exemple. L'objectif est d'écrire la grammaire de base. Au prochain TP vous verez comment vérifier les types :)

2.3 Les structures conditionnelles

Le nom-terminal `IfStmt` doit être utilisé pour les structures conditionnels. Les trois structures suivantes doivent être acceptées par votre grammaire. Concernant la représentation 7, le nombre de section "else if" n'est pas fixé (entre zéro et l'infinie).

Listing 4: If sans {}

```
if (expression)
    statement
```

Listing 5: If avec {}

```
if (expression) {
    block
}
```

Listing 6: If/else

```
if (expression) {
    block
}
else {
    block
}
```

Listing 7: If/else

```
if (expression) {
    block
}
else if (expression) {
    block
}
else {
    block
}
```

2.4 La structure For

Le nom-terminal `ForStmt` doit être utilisé pour les structures for. La structure suivante doit être acceptée par votre grammaire. Les expressions et assignations dans l'entête du for ne sont pas obligatoire, mais les ";" oui.

Listing 8: If/else

```
for (assignation ; expression ; assignation) {
   block
}
```

2.5 Les fonctions anonymes

Vous devrez implementer la structure d'une fonction anonyme qui peut seulement être assignée à une variable. **je vous rapelle que ce n'est pas l'objectif du TP de faire la vérification des types. Donc, pas besoin de vérifier le type de retour. Nous ferons cela au prochain TP :)** Une fonction anonyme en java peut être exprimée de plusieurs manières. Voici quelques exemples:

Listing 9: Fonction anonyme de base

```
() -> {
    block
}

fonc = () Stmt
```

Listing 10: Fonction anonyme avec paramètres

```
fonctionAnonyme = (int a, bool b ) -> {
    block
    return a + 1;
}
fonctionAnonyme2 = (int a) -> Stmt
```

2.6 La structure Switch

Le nom-terminal `SwitchStmt` doit être utilisé pour les structures conditionnelles. La structure suivante doit être acceptée par votre grammaire. Le nombre de "case" est non-défini (entre un et l'infinie) et la section "default" n'est pas obligatoire.

Listing 11: Switch

```
switch ( expression ) {
    case expression : Stmt
    ...
    default : Stmt
}
```

2.7 Priorité des opérations

Les non-terminaux doivent tous terminer par `Expr` pour les expressions que vous allez inventer (*indice* : vous allez en créer plusieurs, ex: `AddExpr`,`MulExpr`, etc.). Une expression est une série d'opération logique ou arithmétique pouvant se résoudre à une valeur. Le langage ne fait pas de différence entre une valeur booléenne et une valeur entière à ce stade-ci.

Pour le moment, dans le code fourni, seul l'addition est implémentée. Vous devez implémenter les autres opérations citées ci-dessous en respectant l'ordre des opérations (de PLUS à MOINS prioritaire) :

1. Parenthèse ("(" et ")")
2. Non logique ("!")
3. Négation ("-")
4. Multiplication "*" et Division "/"
5. Addition ("+") et Soustraction ("-")
6. Comparaison("<",">","<=", ">=", "==", "! =")
7. ET logique ("&&") et OU logique ("||").

2.8 Les nombres réels

Vous devez implémenter le jeton (token) `REAL` représentant les nombres réels. Plusieurs formes de nombres réels sont donnés dans les fichiers de tests.

3 Utilisation du cadriciel et de IntelliJ

Pour davantage de détails concernant le cadriciel, consulter la page du projet sur GitHub:
`https://github.com/Nic007/JavaCC-Template`
Le cadriciel du présent TP a été bâti à partir du cadriciel présent sur GitHub. La structure du projet est la même mais le langage demandé est différent.

3.1 Setup

- Téléchargez l'archive sur Moodle, puis extrayez-la.
- Ouvrez IntelliJ. A la première ouverture :
  - N'importez pas les paramètres.
  - Choisissez votre thème.
  - Décochez la case pour la création d'une entrée de menu.
  - Appuyez sur "Skip remaining and set defaults".
- Ouvre le projet avec "Open".
- Dans la notification qui apparaît au bas de l'écran, suivez les instructions afin d'installer JavaCC, puis redémarrez IntelliJ.
- Ouvrez le fichier `Template.jjt`.

3.2 Utilisation du cadriciel

Vous pouvez désormais apporter vos modification à la grammaire JavaCC. Pour générer et tester l'anaylseur, cherchez dans vos configurations d'exécutions s'il existe "analyzer.Tests" ou "All in template" (à côté du bouton build de IntelliJ a.k.a le marteau vert). S'il n'existe pas, voici les instructions pour créer votre propre configuration.

Si vous avez une erreur avec le JDK introuvable, ouvrez la fenêtre "Project Structure" puis sélectionnez "New → JDK" à côté du champ "Project SDK". Dans la fenêtre qui apparaît, `java-1.8.0-openjdk` devrait être sélectionné. Appuyez sur OK.

4 Comment fonctionne les tests

Les tests ayant comme nom "erreur*.le" ne doivent pas passer. Les autres tests doivent passer au vert. Comment savoir si votre grammaire marche? Il faut voir un arbre cohérent aparaitre lorsque vous cliquez sur le test. C'est très important de comprendre que ce n'est pas parce qu'il y a un crochet vert que votre grammaire est valide. Il est très important de vérifier l'arbre généré et s'assurer qu'il est logique avant la remise.

5 Construire votre grammaire

Voici quelques notations qui vous seront utiles pour l'écriture de votre grammaire.

Pour les tokens/éléments terminaux:

- [...] : set de caractères;
- * : répétition de l'élément précédent de $[0 : \infty]$;
- + : répétition de l'élément précédent de $[1 : \infty]$;
- ? : répétition de l'élément précédent de $[0 : 1]$;
- | : "ou" entre tout ce qu'il y a avant et après le symbole;
- #NOMTOKEN : le nom du token est privé au set de token (et ne peux pas être utilisé en dehors).

Pour les fonctions/éléments non-terminaux:

- * : répétition de l'élément précédent de $[0 : \infty]$;
- + : répétition de l'élément précédent de $[1 : \infty]$;
- [...] : répétition de l'élément entre crochet de $[0 : 1]$;
- | : "ou" entre tout ce qu'il y a avant et après le symbole;
- LOOKAHEAD(N) : regarde les N prochain éléments à ce niveau de l'arbre avant de visiter les noeuds enfants.

6 Test

Il n'y a pas de tests automatiques pour ce laboratoire car vous pouvez choisir vous même les noms dans la grammaire. Cependant, vous pouvez trouver l'impression de vos arbres de parsage en cliquant sur le test une fois "run", en bas de votre écran, une fois votre code exécuté. Vous devez vérifier manuellement les que vos arbres font sens.

Par exemple, le code "c = 1+1;" sera représenté de la manière suivante.

Listing 12: Exemple d'impression d'arbre

```
Program
    Block
        Stmt
            AssignStmt
                Identifier
                Expr
                    AndOrExpr
```

```
ComparisonExpr
    AddExpr
        MultExpr
            NegExpr
                BasicExpr
                    Value
                        IntValue
        MultExpr
            NegExpr
                BasicExpr
                    Value
                        IntValue
```

L'arbre ainsi construit n'est pas très lisible car les étapes intermédiaires de l'arbre sont imprimées. Il est possible de cacher ces parties en utilisant "#void" ou "#customName(condition)".

Listing 13: Exemple de grammaire imprimant "Addition" que s'il y a deux termes.

```
void IntAddExpr() #void :
{
    ( IntMultExpr() ((<PLUS>|<MINUS>) IntMultExpr())* )#Addition(>1)
}
```

En implémentant cette réduction pour l'ensemble des tokens, on peut réduire l'arbre à l'arbre suivant :

Listing 14: Exemple d'impression d'arbre

```
Program
    AssignStmt
        Identifier
        Addition
            IntValue
            IntValue
```

2 points seront attribués à la réduction des arbres. ATTENTION, ces deux exemples ne sont pas forcément à reproduire exactement (notamment au niveau des noms que vous donnez au tokens). A vous d'évaluer la véracité de vos arbres et leur lisibilité.

Vos arbres doivent rester cohérents et différés dans des cas de figurent différents. En particulier, dans le cas des if, faire attention à faire la différence entre les statements de différents blocks.

Les tests "erreur" ne devrait pas passer et apparaitre orange.

7 Barème

Le TP est évalué sur 20 points, répartis comme suit :

- 2 points : production du "while" et "do-while";
- 2 points : production du "if";
- 1 point : production du "for";
- 1 point : production du "switch";
- 6 points : expressions avec tous les opérateurs et l'ordre des opérations;
- 2 points : nombres réels;

- 2 points : réduction des arbres.
- 2 points : fonctions anonymes
- 2 points : tests surprises

Le devoir doit être fait en **binôme**. Remettez sur Moodle une archive nommée *log3210-tp1-matricule1-matricule2.zip* avec uniquement le fichier `Template.jjt` ainsi qu'un fichier `README.md` contenant tous commentaires concernant le projet.

L'échéance pour la remise est le **lundi 7 Février 2022 à 23 h 55**.

Une pénalité de 10 points (50%) s'appliquera par jour de retard. Une pénalité de 4 points (20%) s'appliquera si la remise n'est pas conforme aux exigences (nom du fichier de remise, fichier `Template.jjt` et `README.md` seulement).

Si vous avez des questions, veuillez nous contacter chargé sur le discord du lab dans la chaîne `questions-tp` ou par courriel : gerard.akkerhuis@polymtl.ca et hakim.mektoub@polymtl.ca.