

LOG3210 - Elément de langages et compilateur

TP1 : Grammaire et analyseur syntaxique

Ettore Merlot – Professeur
Quoc-Hao Tran – Chargée de laboratoire

Hiver 2021

1 Objectifs

- Se familiariser avec JavaCC;
- Utiliser 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 opérations 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 structures conditionnelles

Le nom-terminal `IfStmt` doit être utilisé pour les structures conditionnelles. Les trois structures suivantes doivent être acceptées par votre grammaire. Concernant la représentation 4, le nombre de section "else if" n'est pas fixé (entre zéro et l'infinie).

Listing 1: If sans {}

```
if (expression)
    statement
```

Listing 2: If avec {}

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

Listing 3: If/else

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

Listing 4: If/else

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

2.2 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 ("||").

2.3 Déclaration de "class"

Votre grammaire devrait accepter la déclaration d'une "class", avec ses attributs et ses méthodes. On assume qu'il existe seulement 3 types de donnée: **int**, **char**, **boolean**. Les "nested class" sont aussi accepté. Vous pouvez utiliser les non-terminaux comme **ClassDeclaration**, **AttributDeclaration**, **MethodDeclaration**.

REMARQUE

1. Il n'y a pas de **public**, **private**,...
2. Il n'y a pas de **extends**, **implements**,...
3. Il n'y a pas de constructeur...

4. Les méthodes n'ont pas de **return**
5. Les structure 2.1 et 2.2 sont seulement acceptés dans la méthode.

Listing 5: Class declaration

```
class Example {
    class NestedExample {
        classBody
    }
    int a;
    int b, c = 0;
    char methodExample(boolean d, char e) {
        block
    }
}
```

2.4 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

- 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**.

Vous pouvez désormais apporter vos modification à la grammaire JavaCC. Pour générer et tester l'analyseur, faites un "all" dans le panneau Ant. Vous pouvez aussi uniquement générer l'analyseur avec "full-compile" ou exécuter uniquement les tests avec "test".

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.

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.

4 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 peut 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 prochains éléments à ce niveau de l'arbre avant de visiter les noeuds enfants.

5 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 passage 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 6: 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 7: 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 8: 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 aux tokens). A vous d'évaluer la véracité de vos arbres et leur lisibilité.

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

Les tests "erreur" ne devraient pas passer et apparaître orange.

6 Barème

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

- 3 points : production du "if";
- 6 points : expressions avec tous les opérateurs et l'ordre des opérations;
- 7 points : production des "class";
- 2 points : nombres réels;
- 2 points : réduction des arbres.

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 **Dimanche 7 Février 2021 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 me contacter sur moodle ou sur le discord du lab ou par courriel : quoc-hao.tran@polymtl.ca.