

LOG3210 - Éléments de langages et compilateurs

TP1 : Grammaire et analyseur syntaxique

Doriane Olewicki – Chargée de cours
Sara Beddouch – Chargée de laboratoire
Raphaël Tremblay – Chargé de laboratoire
Quentin Guidée – Chargé de laboratoire

Hiver 2023

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 quant 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'elle reconnaisse 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 non-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 Les structures conditionnelles

Le non-terminal `IfStmt` doit être utilisé pour les structures conditionnelles. Les trois structures suivantes doivent être acceptées par votre grammaire. Concernant la représentation 6, le nombre de sections "else if" n'est pas fixé (entre zéro et l'infini).

Listing 3: If sans {}

```
if (expression)  
    statement
```

Listing 4: If avec {}

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

Listing 5: If/else

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

Listing 6: If/else

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

2.3 La structure for

Le non-terminal `ForStmt` doit être utilisé pour la structure `for`. La structure suivante doit être acceptée par votre grammaire. Les expressions et assignations dans l'en-tête du `for` ne sont pas obligatoires, mais les ";;" oui.

Listing 7: For

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

2.4 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'opérations 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 ("||").

2.5 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.

2.6 La structure fonction

Le non-terminal **FunctionStmt** doit être utilisé pour les structures fonction. La structure définie dans la représentation 8 doit être acceptée par votre grammaire. La quantité de paramètres dans l'en-tête du func n'est pas fixe (entre zéro et l'infinie).

De plus, le non-terminal **ReturnStmt** peut être utilisé à la fin des blocs et sous-blocs à l'intérieur de la structure d'une fonction. De cette façon, les fonctions ne peuvent avoir au plus qu'un return par bloc ou sous-bloc et donc il est donc possible qu'il y ait plusieurs returns dans un fonction. Votre grammaire doit accepter ce non-terminal suivi d'une expression optionnelle.

Listing 8: Structure fonction

```
func nom_fonction(a, b, c, ...) {  
    block  
    return expression;  
}
```

3 Utilisation du cadriciel et de IntelliJ

- Téléchargez l'archive sur Moodle, puis extrayez-la.
- Ouvrez IntelliJ. À 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".
- Ouvrez 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 modifications à la grammaire JavaCC. Pour générer et tester l'analyseur, vous pouvez exécuter la cible "All in Template" dans IntelliJ ou faire 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, consultez la page du projet sur GitHub:

<https://github.com/Nic007/JavaCC-Template>

Le cadriciel du présent TP est une version modifiée du cadriciel sur GitHub.

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 Reduction

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

Listing 9: 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 10: 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 11: 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). Vous pouvez vous servir des résultats attendus (**test/PrintTest/expected**) pour évaluer la véracité de vos arbres et leur lisibilité.

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

6 Test

Vous pouvez utiliser les tests automatisés pour valider votre grammaire. Pour chaque test, vous avez une entrée située dans le dossier `test/PrintTest/data` et un résultat attendu avec réduction dans `test/PrintTest/expected`. Vous pouvez trouver l'impression de vos arbres de parsage dans le dossier `test/PrintTest/result`, une fois les tests exécutés.

Il se peut qu'aucun arbre ne s'imprime dans vos résultats, dans ce cas votre grammaire est invalide et aucun arbre n'a pu être généré. Dans ce cas, vous retrouverez une erreur dans la sortie standard du test.

Normalement, tous les tests devraient passer à la completion du TP.

7 Barème

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

Fonctionnalités	
Production du "while" et "do-while"	/3
Production du "if" et "else"	/3
Production du "for"	/2
Expressions avec tous les opérateurs et l'ordre des opérations	/5
Nombres réels	/2
Production du "func" et "return"	/3
Réduction des arbres	/2
Pénalités	
Retard (par jour de retard)	-10
Non-respect des consignes de remise (nom du fichier,...)	-4
Total	/20

8 Remise

Remettez sur Moodle une archive nommée *log3210-tp1-matricule1-matricule2.zip* avec uniquement:

- `Template.jjt`;
- `README.md` (facultatif, à fournir si vous avez des commentaires à ajouter sur votre projet).

L'échéance pour la remise du TP1 est le **5 février 2023 à 23 h 59**.

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).

Le devoir doit être fait en **binôme**. **Les remises individuelles ne sont pas autorisées !** Si vous ne trouvez pas de binôme, vous pouvez regarder sur le Discord du cours. Si vous ne trouvez personne, veuillez nous contacter (minimum 5 jours avant la remise).

Si vous avez des questions, vous pouvez nous contacter sur Discord, sur Moodle ou, pour une urgence, par courriel à

Doriane Olewicki: doriane.olewicki@gmail.com.

Raphaël Tremblay (Gr. 1): raphael-1.tremblay@polymtl.ca

Quentin Guidee (Gr. 2): quentin.guidee@polymtl.ca

Sara Beddouch (Gr. 3): sara.beddouch@polymtl.ca