LOG3210 - Eléments de langages et compilateurs
TP1 : Grammaire et analyseur syntaxique

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

Hiver 2024

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 (`Grammaire.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/SyntaxTest/data` contient des exemples de codes valides et invalides du langage.

2.1 Mots-clés

Tout au long du cours, nous utiliserons certains mots-clés pour définir des catégories de structure dans la grammaire :

- **Statement** : Une instruction du langage (Ex. : assignation, boucle while, définition d'un enum, etc.).
- **Block** : Une suite de statements (0 à ∞).
- **Expression** : Une représentation d'une valeur (Ex. : variable, valeur numérique ou booléenne, expression arithmétique, etc.).
- **Identifier** : Identifiant, par exemple, un nom de variable ou le nom d'un enum. Ils peuvent prendre n'importe quelle forme, ex. : a, myVar, myEnum, etc. Par définition, ces identifiants sont des expressions.

2.2 Les Assignations

Les assignation doivent être supportées par votre grammaire pour tout expression.

Listing 1: Assignation

```
identifier = expression;
```

2.3 Les listes

Les liste [expression, expression, ..., expression] droivent être supportées pour tout nombre d'expression

Listing 2: Liste

```
identifier = [expression, expression, ..., expression];
```

2.4 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 3: Boucle While (deux exemples)

```
while (expression) {
    block
}

while (expression)
    statement
```

Listing 4: Boucle Do-While

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

2.5 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 8, le nombre de sections "else if" n'est pas fixé (entre zéro et l'infinie).

Listing 5: If sans {}

```
if (expression)
   statement
```

Listing 6: If avec {}

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

Listing 7: If/else

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

Listing 8: If/else

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

2.6 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 9: For

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

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'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 ("||").

2.8 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.

2.9 Les enums et la structure switch-case

Le non-terminal `EnumStmt` doit être utilisé pour la structure `enum` et peut contenir une quantité indéfinie d'identifiants.

Listing 10: Définition enum

```
enum MonEnum {
    A, B, ...
}
```

Le non-terminal `SwitchStmt` prend un identifiant en paramètre et contient une suite de non-terminal `CaseStmt` et peut finir avec un `DefaultStmt`. La structure `case` commence avec le jeton `case` suivi d'un identifiant ou d'un entier, puis d'un deux-points. Chaque structure `case` et `default` peut contenir un non-terminal `CaseBlock` constitué d'une suite de statements et peut finir avec un `BreakStmt` représenté par le jeton `break`.

Listing 11: Structure switch-case

```
switch(identifier) {
    case expression:
        block
        break;
    ...
    default:
        block
        break;
}
```

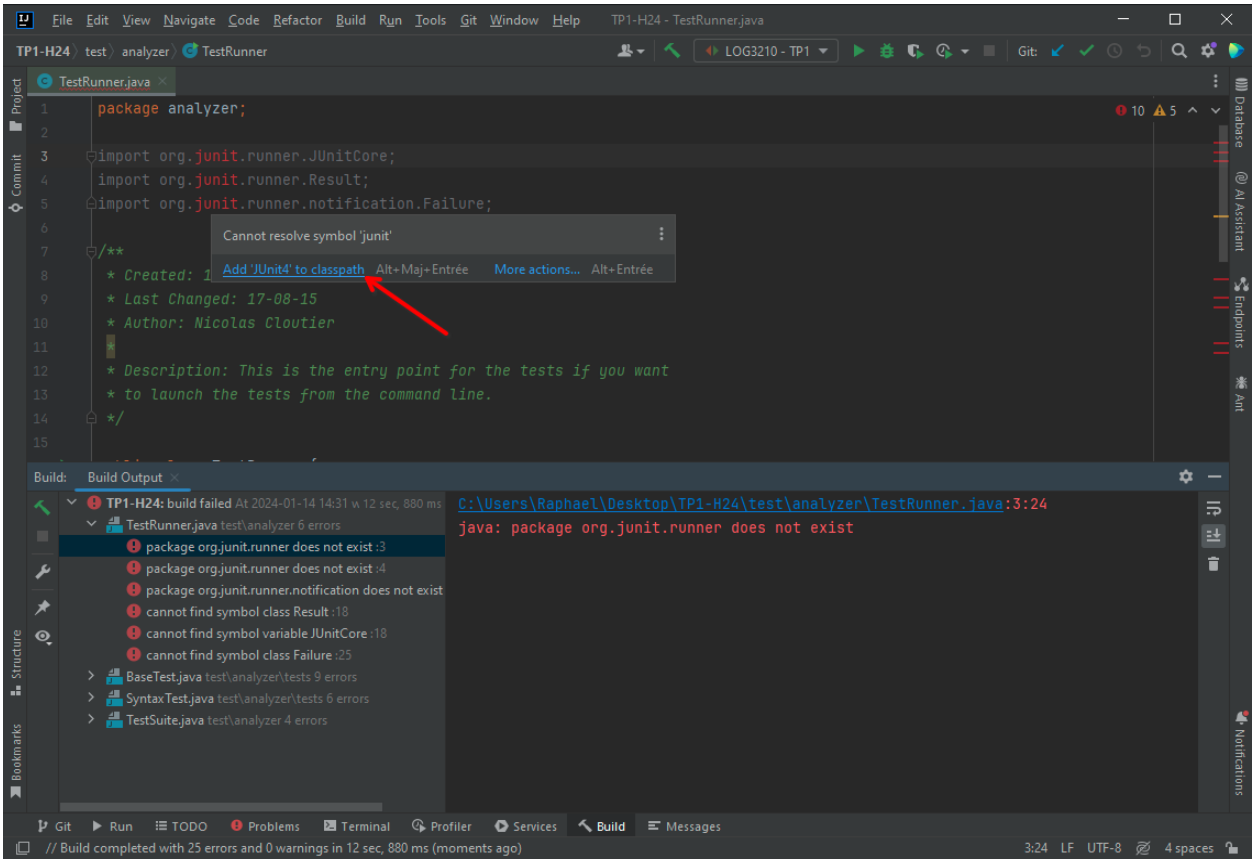
3 Utilisation du cadriciel et de IntelliJ

- 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".
- Installez le plugin JavaCC et Ant proposé par IntelliJ
- Ouvrez le fichier `Grammaire.jjt`.

Vous pouvez désormais apporter vos modification à la grammaire JavaCC. Pour générer et tester l'analyseur, vous pouvez executer la cible "LOG3210 - TP1" dans intelliJ.

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.

Si vous avez une erreur spécifiant des paquets JUnit manquants, placez simplement votre curseur sur le code en rouge, et un menu devrait apparaître vous proposant d'`Add 'JUnit4' to classpath`. Choisissez simplement cette option, puis appuyez sur OK. Par la suite, vous pourrez exécuter la cible "LOG3210 - TP1" et vous devriez voir tous les tests s'exécuter.



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 est une version modifié 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 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.

5 Reduction

Sans réduction, 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
```

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/SyntaxTest/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 Qualité des règles de production

Vos fonctions dans votre grammaire peuvent contenir plusieurs règles de production imbriquées. Dans le cas où notre symbole non-terminal est utilisé seulement dans un autre non-terminal et est assez court, il est intéressant d'utiliser les #libellés pour imbriquer celui-ci.

Listing 15: Exemple de règles de productions imbriquées

```
void FunctionStmt() : {}
{
    <FUNC> Identifier()
    <LPAREN> ((Identifier() [<COMMA>])*) #FunctionParams <RPAREN>
    <LACC> Block() <RACC>
}
```

Dans ce cas, nous imbriquons tous les paramètres de la structure `function` dans le non-terminal `FunctionParams`.

Listing 16: Exemple d'impression d'arbre

```
Program
    FunctionStmt
        Identifier
        FunctionParams
            Identifier
            Identifier
        AssignStmt
            Identifier
            IntValue
```

Deux points seront attribués à la qualité de vos règles de production. Le choix des non-terminaux que vous aurez imbriqués et la quantité de fonctions employées pour construire votre grammaire seront pris en compte. Comme référence, la grammaire solution emploie 25 fonctions

7 Test

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

Dans ce cas votre grammaire est invalide et aucun arbre n'a pu être généré, vous retrouverez une erreur dans la sortie standard du test et dans le fichier resultat.

Tous les tests devraient passer à la completion du TP.

8 Barème

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

Fonctionnalités	
Listes	/1
Production du "while" et "do-while"	/2
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 "switch-case" et "enum"	/3
Qualité des règles de production	/2
Pénalités	
Non-respect des consignes de remise (nom du fichier,...)	-4
Total	**/20**

9 Remise

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

- `Grammaire.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 `4 février 2024 à 23 h 59`.

Pour le premier travail pratique uniquement, nous n'accepterons pas les retards. Une pénalité de 4 points (20%) s'appliquera si la remise n'est pas conforme aux exigences (nom du fichier de remise, fichier `Grammaire.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 avez des soucis avec votre binôme (désinscription au cours, etc.), veuillez nous contacter (minimum 5 jours avant la remise).

Si vous avez des questions, vous pouvez nous contacter sur Discord (dans vos cannaux textuels d'équipe), sur Moodle ou, pour une urgence, par courriel à
Sara Beddouch: sara.beddouch@polymtl.ca
Raphaël Tremblay (Gr. 1): raphael-1.tremblay@polymtl.ca
Quentin Guidee (Gr. 2): quentin.guidee@polymtl.ca