DLP : Développement d'un langage de programmation

Master STL, Université Pierre et Marie Curie

Année 2020–2021

TME 2 – Syntaxe concrète et syntaxe abstraite d'ILP1

Christian Queinnec & Antoine Miné

Objectif global : comprendre la génération de l'arbre syntaxique à partir du source ILP et savoir enrichir la grammaire.

1 Enrichir une grammaire ANTLR 4

Objectif : enrichir la grammaire d'ILP1 avec l'affectation, les boucles et les définitions de fonctions.

Buts :
- comprendre le format de description `.g4` des grammaires ANTLR ;
- savoir ajouter des règles syntaxiques.

1.1 Introduction aux grammaire ANTLR 4

Nous décrivons succinctement quelques aspects importants des grammaires ANTLR 4 en prenant exemple sur la grammaire d'ILP1, contenue dans `ANTLRGrammars/ILPMLGrammar1.g4`. Vous devez avoir ce fichier ouvert en lisant ce texte.

Toute grammaire commence par une ligne :

```
grammar ILPMLgrammar1;
```

qui indique le nom de la grammaire et doit refléter le nom du fichier `.g4` qui la contient, ici `ILPMLgrammar1`. Ce nom sert aussi de préfixe au nom des classes générées par ANTLR 4 à partir de cette grammaire. Nous avons ensuite une entrée :

```
@header {
    package antlr4;
}
```

qui indique que les classes générées appartiennent au *package* `antlr4`. Viennent ensuite un certain nombre de règles syntaxiques de la forme suivante (afin de rendre l'exemple plus pédagogique, l'ordre original des règles du fichier n'a pas été respecté) :

```
1  expr
2  returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
3  :
4    'true' # ConstTrue
5  |
6    intConst=INT # ConstInteger
7  |
8    op=('-' | '!') arg=expr # Unary
9  |
10   'let' vars+=IDENT '=' vals+=expr ('and' vars+=IDENT '=' vals+=expr)*
11   'in' body=expr # Binding
12 |
13   'if' condition=expr 'then' consequence=expr
14   ('else' alternant=expr)? # Alternative
```

— La ligne 1 débute la règle définissant ce qu'est une expression, notée `expr`.
L'objet `expr` est appelé un non-terminal car il est composé de sous-objets (variables, opérateurs, sous-expressions, etc.).

— La ligne 2 précise le type de notre représentation Java pour une expression : c'est un arbre syntaxique abstrait de type `IASTexpression`.
Cette ligne indique que, à tout instant, notre classe *listener*, qui sera appelée par ANTLR au fur et à mesure de l'analyse syntaxique, aura accès au nœud AST en cours de construction dans un attribut nommé `node` et de type `com.paracamplus.ilp1.interfaces.IASTexpression`.

Suit alors une liste de cas, séparés par une barre verticale `|`, correspondant aux différentes manières de créer une expression.

— Ligne 4, `'true'` donne un cas simple : celui d'un mot-clé représentant une constante.
L'indication `# ConstTrue` qui suit précise à ANTLR 4 ce qu'il faut faire quand le mot-clé `true` est reconnu : il faut appeler les méthodes `enterConstTrue` et `exitConstTrue` du *listener* (si `# ConstTrue` n'est pas précisé, alors `enterexpr` et `exitexpr` seront appelées par défaut, ce qui n'est pas très pratique pour distinguer le cas `'true'` des cas qui suivent).

— Ligne 6, `intConst=INT # ConstInteger` donne le cas où un nombre entier est reconnu.
Le sens de `INT` est défini dans une règle lexicale à part, en fin de fichier (il s'agit d'une simple expression régulière).
Comme précédemment, `# ConstInteger` précise que les méthodes `enterConstInteger` et `exitConstInteger` du *listener* seront appelées si un nombre est reconnu.
La partie `intConst=` précise que la chaîne de caractères correspondant au texte reconnu, ici une séquence de chiffres formant un nombre, sera passée à `enterConstInteger` et `exitConstInteger` dans un attribut nommé `intConst`.

— La ligne 8 donne un exemple de règle récursive : une expression peut être une expression unaire, c'est à dire un opérateur, `-` ou `!`, passé au *listener* dans l'attribut `op`, suivi d'une expression, dont le contenu sera passé, sous forme d'`IASTexpression` (c.f., ligne 2) dans l'attribut `arg`.
Notez l'utilisation de `|` pour préciser l'ensemble des opérateurs unaires reconnus.

— La ligne 10 correspond à la définition d'une ou plusieurs variables locales à un bloc :
`let x=expr and y=expr and ... in expr`.
Cet exemple met en évidence l'opérateur de répétition `*`, pour préciser que le motif `and vars=vals` peut être répété.
Les attributs `vars` et `vals` contiennent maintenant des listes, pour stocker toutes les occurrences du motif `and vars=vals`, ce qui est matérialisé par l'utilisation de `vars+=IDENT` et `vals+=expr` au lieu de `vars=IDENT` et `vals=expr`.
`IDENT` correspond à l'expression régulière des identificateurs de variables autorisés.

— La ligne 13 donne la règle du `if then else` et illustre l'utilisation de l'opérateur `?` pour rendre une partie du motif optionnelle, ici la branche `else`.

L'ordre des règles dans la grammaire est important car, en cas d'ambiguïté, la règle apparaissant en premier est choisie en priorité. Ainsi, pour les expressions, nous trouvons le fragment suivant :

```
| arg1=expr op=('*' | '/' | '%') arg2=expr # Binary
| arg1=expr op=('+' | '-') arg2=expr # Binary
```

qui indique que, dans `a+b*c`, `*` est prioritaire sur le `+`, donc l'expression se lit comme `a+(b*c)` et non `(a+b)*c`. Il est également important que `+` et `-` apparaissent dans la même règle pour indiquer qu'ils ont la même priorité ; ainsi `a+b-c` sera lu comme `(a+b)-c` et `a-b+c` sera lu comme `(a-b)+c` : c'est l'ordre d'apparition dans l'expression qui compte ici (nous n'aurions pas le même effet si la règle `+` apparaissait avant celle de `-` dans la grammaire).

1.2 Travail demandé

Nous allons créer une nouvelle grammaire qui étend ILP1 avec les traits de langage suivants :

1. Les affectations :
   ```
   x = 10 + 20
   ```
2. Les boucles `while` :
   ```
   while i < 10 do i = i + 1
   ```
3. La définition de fonctions (en tête de programme) :
   ```
   function f(x,y) x + y
   ```

Ici, nous nous intéressons uniquement à la partie analyse syntaxique, c'est à dire à l'ensemble composé d'une grammaire ANTLR 4 `.g4` et du *listener* appelé par ANTLR 4 à chaque fois qu'une règle syntaxique est reconnue dans le fichier ILP. L'extension à ces constructions de l'arbre syntaxique, de l'interprète et du compilateur n'est pas demandée ici.

Vous effectuerez les étapes suivantes :

1. Créez un *package* `com.paracamplus.ilp1.ilp1tme2.ex1`.
2. Dans ce *package*, créez une copie de `ANTLRGrammars/ILPMLgrammar1.g4` que vous nommerez `ILPMLgrammar1tme2.g4`. L'entrée `grammar` en tête du fichier doit être modifiée pour refléter le nouveau nom de la grammaire : `ILPMLgrammar1tme2`.
3. Ajoutez les règles syntaxiques pour l'affectation, les boucles et la définition de fonction.
   Attention à la position à laquelle vous insérez ces règles : cela influencera la priorité relative des constructions syntaxiques.
4. Lancez ANTLR 4 sur ce fichier pour générer l'analyseur syntaxique Java correspondant. Cela se fait avec un clic droit sur le fichier `.g4`, puis *Run As > Generate ANTLR Recognizer*. Si tout se passe bien, des fichiers `ILPMLgramma1tme2BaseListener.java`, etc. doivent apparaître dans `target/generated-sources/antlr4/antlr4`.
5. Écrivez une classe `ILPMLListener.java` qui implante l'interface `ILPMLgrammar1tme2Listener` générée par ANTLR 4 ; ce fichier reprendra à l'identique toutes les méthodes définies dans `com.paracamplus.ilp1.parser.ilpml.ILPMLListener`, et vous laisserez vides les méthodes correspondant aux nouvelle constructions (puisque nous ne nous intéressons pas à la génération de l'AST dans cet exercice).
   Attention : ce nouveau `ILPMLListener` ne peut pas dériver de celui d'ILP1, donc nous sommes obligés de recopier physiquement les méthodes de l'ancien `ILPMLListener` vers le nouveau (il s'agit d'une limitation actuelle d'ANTLR 4 concernant l'héritage de grammaires).
6. Écrivez une classe `ILPMLParser` qui dérive de la classe de même nom dans ILP1, mais se branche sur la nouvelle grammaire.
7. Écrivez une classe de test `InterpreterTest` et quelques programmes ILP utilisant vos nouveaux traits syntaxiques afin de vérifier que l'étape d'analyse syntaxique fonctionne correctement (il y aura bien sûr un échec à l'interprétation puisque certaines méthodes du *listener* sont encore vides).

2 Parcours de la syntaxe

Objectif : compter, de plusieurs manières, les constantes apparaissant dans un programme ILP1.

Buts :
- comprendre la structure d'AST de ILP1 ;
- comprendre la génération de l'AST par un *listener* ANTLR et savoir le modifier ;
- savoir parcourir l'AST.

Dans cet exercice, nous allons compter le nombre de constantes qui apparaissent dans un programme ILP, en utilisant deux techniques différentes. Rappelons que ANTLR 4 génère à partir d'une grammaire un analyseur syntaxique paramétré par un *listener* précisant à ANTLR 4 ce qu'il faut faire quand une règle syntaxique est reconnue. Dans ILP, le *listener* construit un arbre syntaxique abstrait formé d'instances des classes `ASTexpression`, `ASTprogram`, etc. Pour parcourir le programme et compter les constantes, nous avons donc deux choix :

1. profiter du parcours du programme effectué lors de la création de l'arbre syntaxique ; nous modifierons donc le *listener* pour ajouter le comptage à la volée ;
2. attendre que l'AST soit construit et le parcourir dans une passe ultérieure, dédiée au comptage de constantes, en utilisant l'interface *visiteur* intégrée à l'architecture d'ILP.

Nous allons implanter les deux méthodes.

2.1 Comptage dans le *listener*

Rappelons succinctement le fonctionnement des *listener* ANTLR. Nous utilisons systématiquement dans notre grammaire la syntaxe `# Nom`, qui permet de nommer chaque règle. Nous devons alors fournir à ANTLR 4 un *listener*, c'est à dire une classe qui possède une méthode `enterNom` et une méthode `exitNom` pour chaque règle `Nom`. Quand l'analyseur syntaxique ANTLR détecte que la règle `Nom` peut être appliquée, il appelle ces deux méthodes, en suivant une stratégie de parcours en profondeur de l'arbre syntaxique : il appelle d'abord `enterNom`, puis parcourt la partie du programme reconnue par la règle, et enfin appelle `exitNom`. Notez que le parcours de la partie reconnue par la

règle peut également générer des appels imbriqués à des méthodes `enter` et `exit`. Pour la construction de l'AST, un parcours suffixe est suffisant : un nœud de l'AST est construit après la construction de ses sous-nœuds ; le traitement se fait donc dans les méthodes `exit`, tandis que les méthodes `enter` restent vides.

Travail demandé :

- lisez la classe `ILPMLListener` présente dans le *package* `com.paracamplus.ilp1.parser.ilpml` ; déterminez où sont traitées les constantes ;
- créez un nouveau *package* nommé `com.paracamplus.ilp1.ilp1tme2.ex2` ;
- dans ce *package*, créez une version de `ILPMLListener` qui compte les constantes en incrémentant un attribut à chaque fois qu'une constante est rencontrée lors du parcours du programme ;
  le fichier grammaire `.g4` étant inchangé, il est possible de réutiliser par héritage le *listener* d'origine de ILP1, et de ne redéfinir que les méthodes utiles au comptage ;
- créez une version de `ILPMLParser` et de `InterpreterTest` qui utilisent ce nouveau *listener*, et testez votre implantation ;
- ajoutez dans le fichier `.gitlab-ci.yml` une règle `TME2` pour exécuter cette nouvelle classe de test (en vous inspirant de la règle `TME1` du TME précédent) ; faites un *push* sur le serveur GitLab et vérifiez que les tests fonctionnent sur le serveur.

2.2 Comptage par parcours de l'arbre syntaxique abstrait

Le parcours récursif d'un arbre syntaxique abstrait est une opération très utile, apparaissant à de nombreux endroits dans le code d'ILP. Par conséquent, ILP contient un motif *visiteur* générique qui factorise les nombreux parcours utilisés. Les parcours peuvent différer sur le type d'objet retourné par le parcours (certains n'ont d'ailleurs pas de valeur de retour) et sur les exceptions qui peuvent être lancées lors du parcours. Par ailleurs, certains parcours nécessitent de passer un argument, de type variable, d'un nœud de l'AST à l'autre. Le *visiteur* d'ILP est donc une interface générique, paramétrée par le type de retour, le type d'argument et le type d'exception :

```
1 public interface IASTvisitor<Result, Data, Anomaly extends Throwable> {
2     Result visit(IASTalternative iast, Data data) throws Anomaly;
3     Result visit(IASTbinaryOperation iast, Data data) throws Anomaly;
4     ...
5 }
```

Travail demandé :

- lisez l'interface `IASTvisitor` présente dans le *package* `com.paracamplus.ilp1.interfaces` ;
  déterminez où sont traitées les constantes ;
- lisez la classe `Interpreter` dans `com.paracamplus.ilp1.interpreter` ; il s'agit d'un exemple de visiteur implantant l'interface `IASTvisitor` ;
- dans le *package* `com.paracamplus.ilp1.ilp1tme2.ex2`, créez une classe `CountConstants` qui implante `IASTvisitor` et renvoie le nombre de constantes dans l'AST ;
  quelles classes choisir pour `Result`, `Data` et `Anomaly` ?
  prenez bien garde à effectuer un parcours récursif complet, sans oublier de nœud : il est de la responsabilité de chaque méthode `visit(...)` d'appeler récursivement le visiteur sur chacun de ses sous-nœuds ;
- modifiez la version de `ILPMLParser` développée à la question précédente pour appeler le *visiteur* sur l'AST juste après sa construction ;
- testez votre implantation en local sur Eclipse, puis sur le serveur GitLab après un *push*.

3 Rendu

Comme pour le TME précédent, vous effectuerez un rendu en vous assurant que tout le code développé a été envoyé sur le serveur GitLab (*push*), que les tests d'intégration continue fonctionnent, et enfin en ajoutant un tag « rendu-initial-tme2 » en fin de séance, puis un tag « rendu-final-tme2 » quand le TME est finalisé.