Université de Namur

IHDCB331 - Algorithmique II

Théorie des langages : syntaxe et sémantique - Compilateur

Les développeurs savent pourquoi !

Groupe 4

Etudiants :
Hadrien Bailly
Anthony Di Stasio
Benoît Duvivier

Professeur :
Pierre-Yves Schobbens
Assistant :
James Ortiz

13 Mai, 2021

Table des matières

Chapitre 1

Introduction

Dans le cadre du cours de Théorie des langages : Syntaxe et Sémantique, le développement d'un projet Java nous est demandé. En effet, celui-ci doit nous permettre d'utiliser les notions vues en cours, en réalisant un compilateur. Celui-ci doit pouvoir convertir un langage PILS en un langage NBC dans le but de contrôler un robot Lego. Les outils à utiliser sont ANTLR4 pour la grammaire, GitHub pour la plateforme d'échange de code et Maven pour la gestion des plugins.

Démarche générale

Tout au long du projet, un chef de projet s'est désigné assez naturellement. En effet, Hadrien a rapidement pris ce rôle afin de faciliter le développement.

La méthodologie était de penser à une architecture et de développer le squelette pour pouvoir ensuite diviser les tâches au sein du groupe. Pour ce faire, nous avons utilisé des fonctionnalités offertes par GitHub. Nous avons créé des tickets pour les tâches à effectuer, des *milestones* pour correspondre aux échéances données. Mais le plus important, un système de peer-reviewing grâce aux pull requests. De cette manière, nous pouvions nous rapprocher au maximum d'une situation professionnelle réelle.

Pour avoir un aperçu de l'avancement du projet, nous avions régulièrement des réunions. Grâce à celles-ci, nous pouvions analyser les problèmes de chacun et essayer, au mieux, de les résoudre pour débloquer la situation.

1.1 Calendrier

Date	**Délivrables**
07 mars 2021	Syntaxe 1-6
11 avril 2021	Syntaxe et sémantique 1-11
11 mai 2021	Remise du code (GitHub)
13 mai 2021	Remise du rapport (WebCampus)

1.2 Architecture du compilateur

1.2.1 Structure

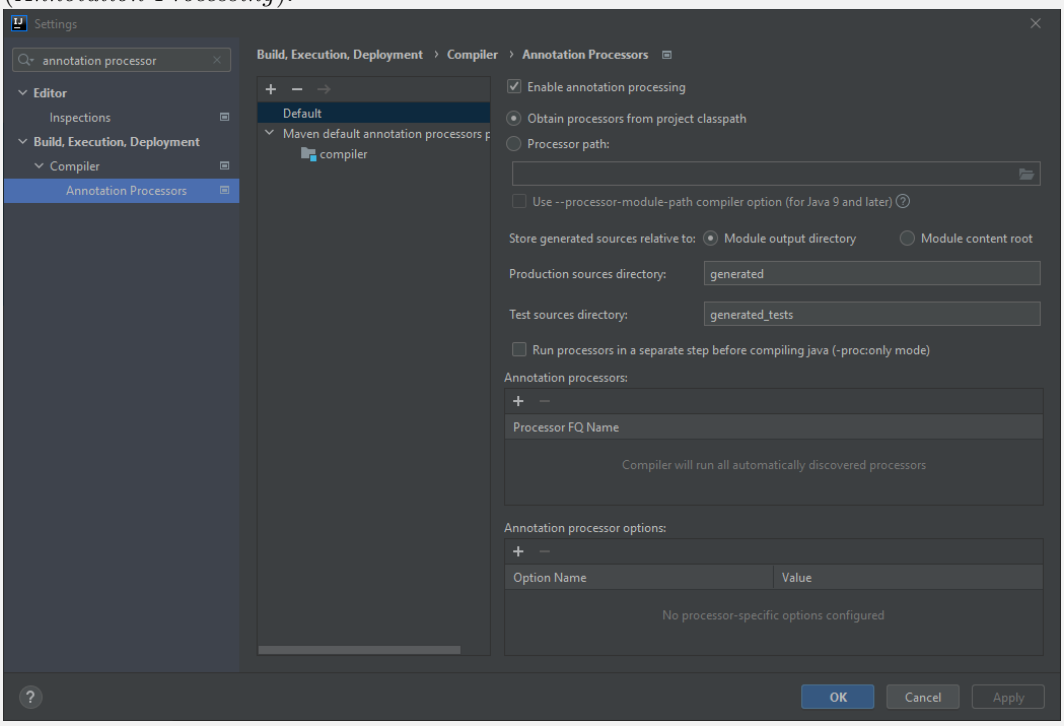
Le compilateur est divisé en plusieurs parties :

- **Main** - *Contient la méthode main permettant de traiter les fichiers .wld et .b314 et au final, de les convertir en langage NBC.*
- **PILS** - *Contient les classes permettant de modéliser le langage PILS.*
- **NBC** - *Contient les classes permettant de modéliser le langage NBC.*
- **Arena** - *Contient la classe Arena qui permet de vérifier si la map est valide.*
- **Listeners** - *Contient les méthodes permettant d'observer les éléments présents dans les fichiers reçus (b314 et wld). Celles-ci servent à vérifier la syntaxe, la sémantique et finalement, créer les objets du langage PILS. Les méthodes se déclenchent automatiquement à la lecture du fichier.*
- **Visitors** - *Contrairement aux listeners, les visitors se déclenchent manuellement pour lire certaines parties de l'arbre.*
- **Mappers** - *Contient les classes permettant de convertir un langage source PILS vers un langage cible NBC.*

1.2.2 Lombok

Pour faciliter le développement, nous avons décidé d'utiliser le plug-in Lombok. C'est un plugin Java permettant de réduire du code *boilerplate* que nous avons l'habitude d'écrire lorsque nous créons des objets modèles ou de données, par exemple. Grâce aux annotations qu'il fournit, nous pouvons générer des méthodes et ainsi, ne pas devoir les écrire, ni même les voir dans le code. Sa plus simple annotation étant *@Data*, celle-ci permet justement de créer des getters et setters pour la classe où elle est appliquée.

L'utilisation de Lombok dans IntelliJ nécessite l'activation du traitement des annotations (*Annotation Processing*).



Pour plus d'informations sur l'utilisation et la configuration de Lombok, un article sur l'utilisation de Lombok dans un IDE est disponible sur le site Baeldung.

Première partie

Analyse

Chapitre 2

Grammaire

Ce projet utilise la librairie ANTLR4, le plugin Maven antlr4-maven-plugin et l'extension IntelliJ ANTLR v4.

2.1 Introduction

ANTLR4 permet de créer des règles de grammaire avec plus ou moins de modularité.
Nous avons identifié les règles suivantes comme les plus importantes :

- Une grammaire est composées de règles (*rule*) et de mots (*token*).
- L'ensemble des règles et des mots doivent être décrits dans des fichiers à l'extension **.g4**.
- Les règles et les reconnaissances de mots s'effectuent selon la règle "*longest match first*" (la plus longue règle applicable est sélectionnée). Lorsque deux règles sont en conflit, c'est alors l'ordre d'énonciation qui l'emporte.
- Aucune règle ne peut s'achever sur un mot vide.
- Il est possible de répartir ces règles et mots dans différents fichiers. Il est alors possible de les relier via l'instruction "import". Les règles sont alors transposées en début de fichier à la génération (ce qui impacte dès lors l'ordre de résolution).

> L'import de fichier répond à une règle différente selon que la grammaire est traitée par l'extension IntelliJ et le plugin Maven : l'extension requérait que l'ensemble des fichiers soient présents sous le même répertoire, tandis que le plugin suit une structure en deux répertoires.
>
> **ANTLR v4 Maven plugin - Default directories**
>
> ```
> src/main/
>      |
>      +--- antlr4/... .g4 files organized in the required package structure
>             |
>             +--- imports/  .g4 files that are imported by other grammars.
> ```

2.2 Structure

Forts de ces règles, nous avons décidé de suivre la structure suivante pour l'organisation de nos règles.

- Dans le dossier principal se trouve le fichier *Grammar.g4*, qui contient à la règle générale (*root*) ainsi que les deux règles spécifiant les règles gouvernant les deux types de fichiers supportés par le langage PILS :
  - Les fichiers contenant la description des stratégies (**.b314**)
  - Les fichiers contenant les élaborations de monde (**.wld**)
- Dans le dossier "import" se trouvent les différents fichiers de règles internes, répartis en sections logiques :

Tokens

- **Action** - *La liste des mots représentant une action NEXT.*
- **Direction** - *La liste des mots représentant une direction.*
- **Items** - *La liste des objets.*
- **KeyWords** - *La liste des mots-clés réservés du langage.*
- **Variables** - *La liste des fragments et mots de base : types, identifiants, nombres.*
- **Words** - *Un fichier permettant de lister et de référencer tous les autres fichiers de vocabulaire en un unique import.*

Rules

- **Clauses** - *Les règles gouvernant les clauses : clause When et clause Default.*
- **Declarations** - *Les règles décrivant les déclarations de variables et de fonctions.*
- **Expressions** - *Les règles de rédaction des expressions.*
- **Imports** - *Les règles pour l'import des fichiers .wld au sein d'une stratégie.*
- **Instructions** - *Les règles indiquant la structure des instructions du langage PILS.*
- **Types** - *Les règles de déclaration d'un type (en lien avec les variables et fonctions).*

2.3 Contenu

La grammaire est issue du document de spécifications distribué par M. Ortiz Vega : nous en avons tout d'abord retranscrit les règles directement dans les fichiers .g4, en adaptant la syntaxe pour qu'elle corresponde au prescrit de l'outil ANTLR4.
Nous avons ensuite procédé à deux adaptations :

- Traitement d'un problème de récursion à gauche.
- Extraction de la règle expression.

2.3.1 Récursion à gauche

ANTLR4 est capable de gérer seul les cas de récursion à gauche *simple* : lorsqu'une règle s'auto-référence en tant que première règle. En revanche, cet outil n'est pas capable de traiter les cas de récursions sur plusieurs niveaux : lorsque la récursion à gauche s'effectue sur plusieurs niveaux, de sorte qu'elle n'est pas directement visible sur une règle bien particulière, ANTLR n'est pas à même de résoudre le problème mais parvient néanmoins à le signaler.

Dans le cadre de ce projet, une récursion à gauche complexe s'est déclarée au niveau des expressions, au sein des deux règles suivantes :

```
ExprD ::= ExprEnt
        | ExprBool
        | ExprCase
        | ExprG
        | Id(ExprD(,ExprD)*)?
        | (ExprD)
```

```
ExprEnt ::= Entier
          | latitude | longitude | grid size
          | (map|radio|ammo|fruits|soda)
              count
          | life
          | ExprD + ExprD
          | ExprD - ExprD
          | ExprD * ExprD
          | ExprD / ExprD
          | ExprD % ExprD
```

ExprD peut en effet dépendre de **ExprEnt** pour la réalisation de l'une de ses alternatives. Or, **ExprEnt** dépend lui-même de **ExprD** pour les cas d'opérations mathématiques. Dans ce cas, le parseur d'ANTLR4 n'est pas en mesure d'arrêter l'exécution d'une règle correctement et le compilateur lève donc une exception.

Pour résoudre ce problème, nous avons retourné la forme de ces deux règles : plutôt que d'avoir une expression droite dont la valeur est une entité, qui peut être soit une (sous-)expression soit une valeur, nous avons créé une règle qui dit d'une expression qu'elle est composée directement soit d'une opération, soit d'une valeur.

```
expression: expression operation expression | value;
```

```
operation: op=( '+' | '-' | '*' | '/' | '%' | AND | OR | '>' | '<' | '=' );
```

```
value: integer | bool | square | reference | '(' expression ')';
```

De cette manière, nous rétablissons une forme de récursivité à gauche plus *simple*, qui peut être résolue par ANTLR4 : seule **expression** est en effet en récursion sur **expression**.

2.3.2 Règle Expression

Une fois la récursivité à gauche traitée, nous avons ajouté une étape qui peut sembler inutile au premier abord :

```
expression: subExpression;
```

```
subExpression: subExpression operation subExpression | value;
```

L'introduction de cette notion de sous-expression n'a pas de plus-value quant au traitement des règles par ANTLR4, puisqu'elle n'a qu'une seule alternative. En revanche, elle simplifie le traitement des expressions plus tard lors de leur visite, en donnant un point d'accès clair :

- Lorsque l'on entre dans la règle **expression**, cela sera nécessairement la règle d'entrée.
- Lorsque l'on entre dans la règle **subExpression**, nous serons nécessairement dans une phase de récursion.

> Nous reviendrons sur ce point lorsque nous aborderons la visite et la validation des expressions.

2.4 Pipeline

Pour la phase de test de notre grammaire, nous avons rencontré des difficultés avec l'exécution des tests sur Jenkins : lorsque nous travaillions pour la première échéance, les tests semblaient ne retourner aucune erreur lors des phases de build.

Lorsque nous lancions une phase depuis Jenkins, le message suivant était affiché :

Jenkins Build Output

```
Running as SYSTEM
[EnvInject] - Loading node environment variables.
Building in workspace /var/jenkins_home/workspace/2021_IHDCB332_G04_echeance2
[2021_IHDCB332_G04_echeance2] $ /bin/bash /tmp/jenkins692289231990837973.sh
Cloning into '2021_SyntaxeSemantique_Prof'...
Cloning into '2021_IHDCB332_G04'...
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Pils Compiler 1.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ compiler ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.374 s
[INFO] Finished at: 2021-04-02T10:44:54+00:00
[INFO] Final Memory: 9M/304M
[INFO] ------------------------------------------------------------------------
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Pils Compiler 1.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- antlr4-maven-plugin:4.9.1:antlr4 (default) @ compiler ---
[INFO] ANTLR 4: Processing source directory
     /var/jenkins_home/workspace/2021_IHDCB332_G04_echeance2/2021_IHDCB332_G04/src/main/antlr4
[INFO] Processing grammar: Grammar.g4
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ compiler ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.2:compile (default-compile) @ compiler ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 165 source files to
     /var/jenkins_home/workspace/2021_IHDCB332_G04_echeance2/2021_IHDCB332_G04/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 10.660 s
[INFO] Finished at: 2021-04-02T10:45:06+00:00
[INFO] Final Memory: 32M/636M
[INFO] ------------------------------------------------------------------------
Recording test results
None of the test reports contained any result
[WS-CLEANUP] Deleting project workspace...[WS-CLEANUP] done
Finished: SUCCESS
```

Plusieurs éléments semblaient indiquer que tout s'effectuait correctement de notre point de vue :

- Notre grammaire était correctement traitée par le plugin Maven.

```
[INFO] ANTLR 4: Processing source directory /var/jenkins_home/workspace
     /2021_IHDCB332_G04_echeance2/2021_IHDCB332_G04/src/main/antlr4
[INFO] Processing grammar: Grammar.g4
```

- La phase Maven build affichait un message de réussite.

```
[INFO] BUILD SUCCESS
```

- Le job Jenkins affichait également un message de réussite.

```
Finished: SUCCESS
```

De même, tous les tests que nous avions rédigés de notre côté se déroulaient sans erreur (nous en avions 144 en date du 21/02).

Commit "Add files via upload" du 21/02/2021 @ 18 :38
SHA :2816b9f9903e1a7435367b74b9abea53ed942f52

Le message *None of the test reports contained any result* nous a bien sûr semblé étrange mais dans la mesure où la grammaire était calquée quasi à l'identique sur la spécification qui nous avait été donnée et que nos tests se déclenchaient bien via Maven et s'effectuaient sans erreur, ce n'est que lors de la 2e échéance que le doute raisonnable s'est dissipé.

Nous en avons alors avisé M. Ortiz Vega et avons tenté de remédier à la situation au plus vite :

- Nous avons reçu les accès à un second environnement Jenkins sur lequel nous avons pu effectuer des tests.
- Nous avons procédé par étapes successives en ajoutant au fur et à mesure les développements que nous avions faits.

Nous avons ainsi exploré plusieurs pistes :

- Un problème au niveau des versions des librairies ANTLR 4 (que nous avions augmentée à la version 4.9.1), Log4j (que nous avions changée au profit de Logback), ou à la présence de Lombok.
- Un problème au niveau de l'exécution des scripts Python présents dans le *repository* et servant à la génération automatique des tests.

En fin de compte, la raison principale était tout autre : dans le fichier pom.xml donnant les instructions de build, une référence en dur vers la classe *main* était présente.

Maven POM.xml excerpt

```
<!-- Package the jar and libs -->
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>${maven-assembly-plugin.version}</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <mainClass>be.unamur.info.demo.compiler.main.Main</mainClass>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <!-- this is used for inheritance merges -->
            <id>make-assembly</id>
            <!-- bind to the packaging phase -->
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Or, nous avions déplacé la classe Main lorsque nous avons structuré notre projet. Ceci a dès lors conduit à l'arrêt de la phase d'assemblage et donc de l'exécution des tests.

Aucun message d'erreur n'était cependant produit de notre côté en local, ni sur le rapport de *build* de Jenkins.

Le rétablissement du package main avec la classe Main le 6 avril à 23h16 nous a alors permis de reprendre les tests et de corriger nos erreurs avant la remise de la seconde échéance.

Chapitre 3

Analyse sémantique

3.1 Présentation du programme PILS

PILS est le langage source de notre compilateur. Nous avons décidé de construire le programme en mémoire dans le but de faciliter son utilisation dans le code. La structure est expliquée ci-dessous.

3.1.1 Structure

Dans la plupart des classes représentant le langage, se trouven les méthodes suivantes :

- **from** - *Cette méthode reçoit un contexte venant du GrammarParser généré par ANTLR et permet de construire un objet en mémoire.*
- **toString** - *Cette méthode permet de recréer le code PILS sous forme de String.*
- **isValid** - *Cette méthode vérifie, comme son nom l'indique, si les attributs de cette classe sont valides.*

Ensuite, la structure est divisée en plusieurs packages :

Program

Le package *Program* contient les classes permettant de construire complètement en mémoire une stratégie ou un monde, nommées *Strategy* et *World*. Ensuite, étant donné que des parties de la grammaire sont dupliquées entre le monde et la stratégie, une classe abstraite nommée *Program* est utilisée. Elle devient la classe parente des deux autres. Les classes de ce package utilisent toutes les autres classes des autres packages pour pouvoir construire ce programme.

Keywords

Le package *Keywords* contient tous les mots-clés utilisés dans le jeu. Ceux-ci sont représentés à l'aide d'énumérations. Celles-ci implémentent l'interface *Reserved* qui leur permet de récupérer tous les keywords d'une énumération, etc. Mais aussi, d'un autre coté, de vérifier si une valeur passée en paramètre est un mot-clé.

Exceptions

Le package *Exceptions* contient toutes les exceptions que nous avons créées pour les rendre plus explicite lorsqu'elles sont déclenchées.

Import

Le package *Import* contient la seule et unique classe portant le même nom que le package. Elle permet donc de créer un objet de type *Import* possédant le nom du fichier.

Statements

Le package *Statements* contient une interface et 6 classes. Chaque classe implémente *Statement*. Ces 6 classes gèrent les instructions *compute*, *set*, *if*, *while*, *next* et *skip*.

Declarations

Le package *Declarations* contient les classes permettant la déclaration des variables et fonctions. Celles-ci sont typés grâce à une énumération nommée *Type*. Les classes *Variable* et *Function* étendent la classe abstraite *Symbol* leur permettant d'avoir un *id*, un *name* et un *level* qui seront utilisés dans la table de symboles (voir 3.3.3).

Values

Le package *Values* contient les classes permettant de représenter la valeur d'une variable, d'un retour de fonction et bien d'autres. Celle-ci est typée grâce à la classe *Type* du package *Declarations*.

Expressions

Le package *Expressions* possède les classes permettant de construire une expression en découpant en *ExpressionLeaf* et *ExpressionNode*. Les expressions sont expliquées plus en détails dans le point 3.4.

Actions

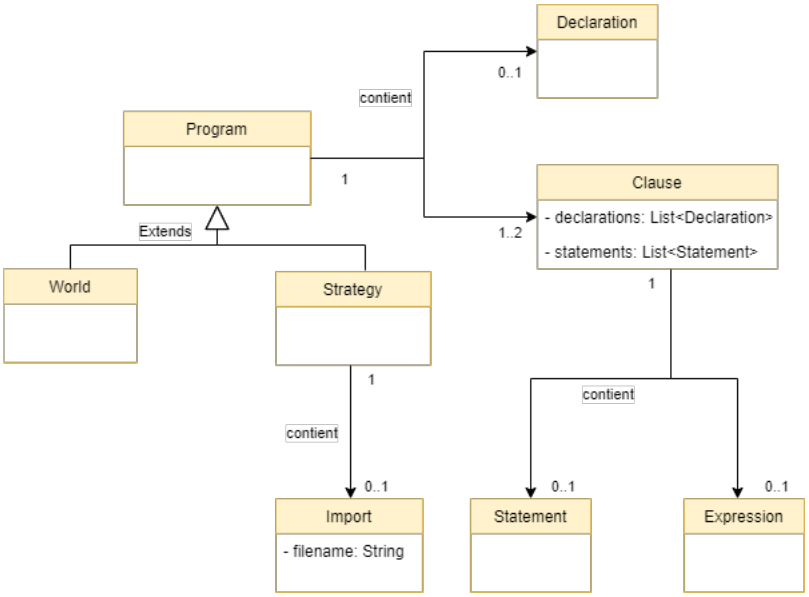
Le package *Actions* contient les classes permettant de gérer les actions *move*, *use*, *shoot* et *do nothing*. Ces actions sont utilisées par la classe *StatementNext*.

Clauses

Le package *Clauses* contient les classes permettant de construire les objets *ClauseDefault* et *ClauseWhen*.

3.2 Modélisation avec un diagramme de classe UML

Voici une représentation de la modélisation des classes du langage PILS que nous avons écrit. Celui-ci n'est pas complet et contient seulement les classes principales.



3.3 Listeners

Pour pouvoir parser les fichiers b314 et wld, nous devons implémenter l'observer pattern. En effet, grâce à une classe générée par *ANTLR*, nommée *GrammarBaseListener*, nous pouvons étendre notre classe pour implémenter les méthodes *enter* et *exit* de chaque partie de notre grammaire. Celles-ci vont donc, comme le nom du pattern l'indique, observer les évènements.

3.3.1 Instructions enter et exit

Comme dit auparavant, la classe générée *GrammarBaseListener* nous permet de réécrire des méthodes telles que *enterStrategy*, *exitStrategy* ou encore *enterWorld* et *exitWorld*. Grâce à ces méthodes, nous allons pouvoir parser chaque partie de la grammaire en créant un objet, par exemple, de type *Strategy* et vérifier la sémantique de celui-ci. Autrement dit, nous parcourons donc l'arbre avec ces méthodes.

C'est aussi dans ces méthodes que l'objet *context* est initialisé et complété. Celui-ci contient, notamment, la table de symboles (voir 3.3.3) mais aussi la variable d'environnement *arena* (voir 3.6.1).

3.3.2 Visitors

Nous faisons aussi appel au *Visitor pattern* car, à certains endroits, nous avons besoin de récupérer plus d'informations à propos d'une partie de la grammaire. Pour ce faire, nous passons un objet en paramètre à la classe *GrammarVisitor* qui est générée par *ANTLR*. Ensuite, nous pouvons parser la partie de la grammaire et en retourner un objet complet et sémantiquement vérifié.

3.3.3 Table des symboles

L'objectif de la table des symboles est de maintenir et de mettre à jour, au fur et à mesure de la lecture du programme PILS, un inventaire des symboles (variables et fonctions) qui sont déclarés au sein de celui-ci.

Lors de l'élaboration de cette table, nous avons pris le parti de diviser celle-ci en deux sous tables qui se présentent sous la forme de deux Hashmaps. Cette façon de procéder nous permet alors de stocker, d'une part, les déclarations des différentes variables, et d'autre part, les déclarations des différentes fonctions. Chacune de ces Hashmaps contient ainsi une clé (représentée par le nom du symbole), ainsi que sa définition.

Lors de la lecture du programme, un objet de type Variable/Function est alors créé, et ses attributs (e.g. type, nom, portée) sont initialisés grâce aux informations récoltées lors de la lecture de sa déclaration. Une fois reconstitué, l'objet est alors ajouté dans la Hashmap correspondante.

La déclaration des fonctions est un cas un peu particulier, car celle-ci contient, en son sein, d'autres déclarations, à savoir celle des éventuelles variables locales, et des éventuels arguments. Par conséquent, lors de la déclaration d'une variable locale/d'un argument, celle-ci est alors ajoutée à la fois dans la table des symboles des variables sous la forme d'une clé et de sa définition, mais également dans la table des symboles des fonctions sous la forme d'un attribut de l'objet Function correspondant à la fonction dans laquelle celles-ci sont déclarées.

C'est à partir de cette table des symboles qu'il nous est alors possible d'effectuer l'ensemble des vérifications concernant le respect des règles du langage PILS, telles que :

- Le respect de la portée des variables
- Le respect de l'unicité du nom d'une variable/fonction du même contexte (cf. 3.5)
- L'absence de conflit entre le nom d'une variable/fonction et un nom réservé
- La cohérence des types, notamment au niveau des opérations, et de la correspondance entre le type d'une fonction et celui de sa valeur de retour.

En effet, cette table contenant l'ensemble des informations correspondantes aux différentes déclarations, il devient alors facile de récupérer une information précise dont le visitor a besoin pour effectuer ces différentes vérifications.

3.4 Expressions

La visite des expressions a soulevé la problématique de l'ordre de préséance des opérateurs : une expression se lisant simplement de gauche à droite, les opérations de calcul arrivent en ordre chronologique et non pas de priorité. Ceci a eu deux conséquences :

- À chaque lecture d'une nouvelle opération, il faut potentiellement réorganiser l'expression selon son degré de priorité.
- Il n'est pas possible de déterminer la valeur de validité d'une expression avant de l'avoir lue dans son entièreté.

3.4.1 Ordre de priorité

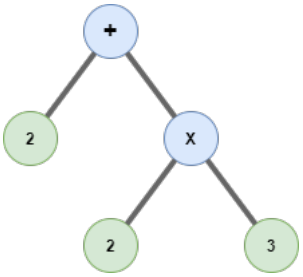
Comme expliqué lors de la présentation du programme PILS, une expression se présente sous la forme d'un arbre, dont les branches sont des valeurs et les noeuds les opérateurs qui les relient.

Soit l'expression $2 + 2 \times 3$.

D'après le degré de priorité des opérateurs *plus* et $fois$, ce calcul s'effectue dans l'ordre suivant :

1. $2 \times 3 = 6$
2. $2 + 6 = 8$

L'arbre formé par cette expression est ainsi le suivant



- L'opération 2×3 ne dépend que des deux valeurs entières 2 et 3.
- L'opération $2 + (6)$ dépend du résultat de l'opération de plus haute priorité 2×3

Si l'expression se présente sous une forme simple, où les opérations sont triées par ordre de priorité, reconstruire l'arbre est "facile" : il suffit d'ajouter des noeuds dont l'une des branches utilise le résultat de l'ensemble des opérations précédentes. Cependant, un tel raisonnement emprunte un postulat d'ordre qui n'est en rien garanti : *quid* si l'expression arrive en ordre inverse de priorité $(3 \times 2 + 2)$, ou pire, dans le désordre $(1 + 2 \times 3 + 1)$?

> Il n'est pas possible de faire en sorte que l'expression soit triée (c'est-à-dire orientée uniquement d'un côté de l'arbre, de manière à ce qu'on puisse facilement ajouter de nouveaux éléments) : si plusieurs opérations de même priorité doivent être effectuées (par exemple, dans $2 \times 2 + 2 \times 2$), il faut nécessairement équilibrer l'arbre.

De la priorité des opérateurs

Avant toute choses, nous avons établi une liste de priorité des opérateurs. Nous avons ainsi dressé la pyramide suivante :

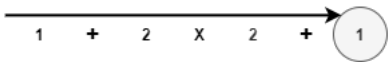
1. Les opérateurs mathématiques de multiplication (MUL), division (DIV) et modulo (MOD).
2. Les opérateurs mathématiques d'addition (ADD) et de soustraction (SUB).
3. Les opérateurs de comparaison stricte "plus grand que" (GRT) et "plus petit que" (LSS).
4. L'opérateur d'équivalence (EQ).
5. L'opérateur booléen de conjonction (AND).
6. L'opérateur booléen de disjonction (OR).

On considérera ensuite qu'un opérateur à gauche a préséance sur un second si celui-ci est d'ordre supérieur ou égal à celui-ci.

Des rotations

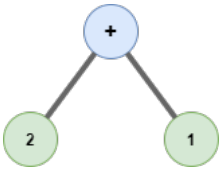
Pour parvenir à construire l'arbre représentant l'expression en parcourant celle-ci de manière progressive, nous procédons par *rotation*.
Soit l'expression suivante $1 \times 1 + 2 \times 1 + 1$.

- Cette expression se construit au moyen du visiteur ExpressionVisitor.
- Le visiteur lit l'expression entièrement, en partant de la gauche.



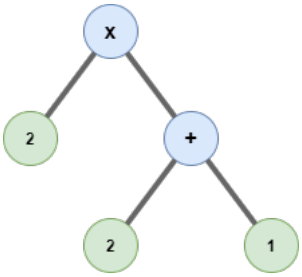
- Une fois arrivé au dernière élément de l'expression, il commence la construction de l'expression, en identifiant la première sous-expression "$2 + 1$".

  Il produit alors un premier arbre simple.



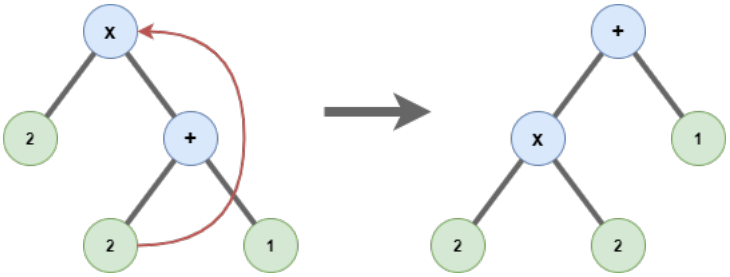
- Il poursuit sa lecture et lit un nouveau membre : "2×"

Il ajoute alors simplement ce nouvel élément en haut de l'arbre.



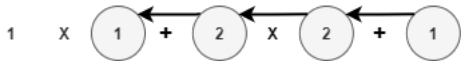
> L'ordre de priorité est alors rompu : prise telle quelle, l'opération présenterait un résultat de $2 \times (2 + 1) = 6$ au lieu de 5.

Il faut dès lors restructurer l'arbre pour tenir compte de la priorité. On compare le nouvel opérateur avec l'opérateur précédent : si celui-ci est plus prioritaire ou équivalente, on effectue une rotation à gauche.

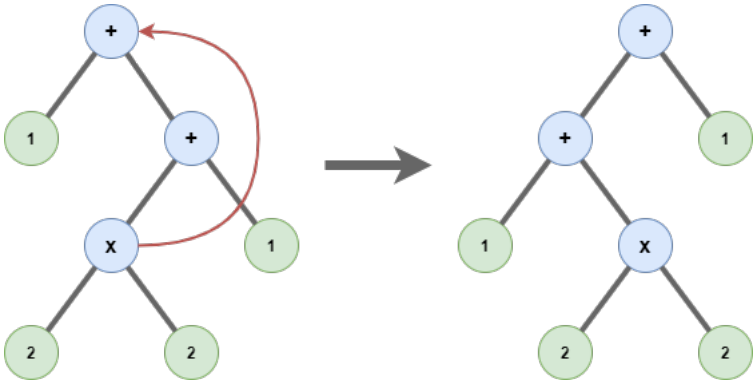


L'ordre est alors restauré : $(2 \times 2) + 1 = 5$

- Le visiteur lit ensuite le membre "1 + ".



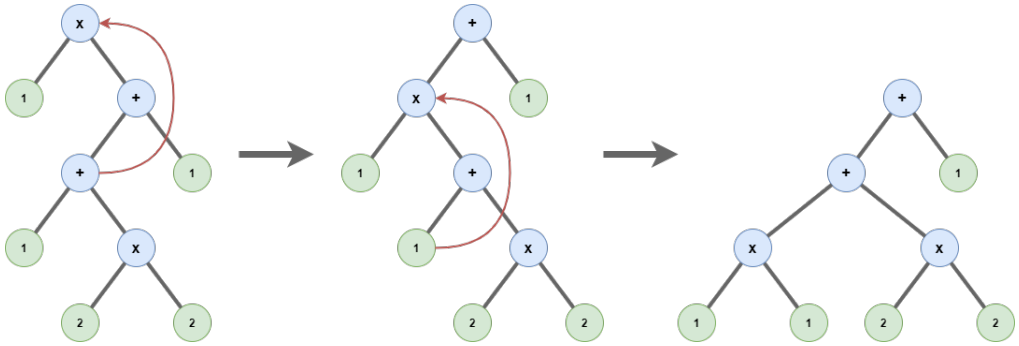
On effectue une nouvelle rotation à gauche pour garantir un traitement de l'expression de gauche à droite à priorité égale.



L'opérateur ADD étant strictement non prioritaire par rapport à l'opérateur MUL, les rotations s'arrêtent.

- Enfin, le visiteur accède au dernier élément "1×".

Ici, l'opérateur est toujours de priorité supérieure ou égale. Dès lors, on effectue encore des rotations : une première fois face à l'opérateur ADD, puis une seconde avec l'opérateur ADD suivant. Une fois qu'il n'y a plus que des feuilles, il n'est plus possible d'effectuer de nouvelle rotation.



3.4.2 Calcul de la validité

Une expression peut combiner différent types de valeurs : booléen, entier ou case. Ces valeurs appartiennent à des domaines de valeurs distincts et ne peuvent normalement pas être combinées entre elles.

- Les valeurs entières sont définies sur le domaine des entiers $\mathbb{N}$ et autorisent les opérations d'addition (+), soustraction (-), multiplication (*), division (/), modulo (%), comparaison supérieure (>) et inférieure (<) et équivalence (=).
- Les valeurs booléennes sont définies sur le domaine $\{true, false\}$ et admettent les opérations de conjonction (and), disjonction (or) et équivalence (=).
- Les valeurs de case sont définies sur le domaine {DIRT, ROCK, VINES, ZOMBIE, PLAYER, ENEMY, MAP, RADAR, RADIO, AMMO, FRUITS, SODA, GRAAL} et admettent uniquement l'opérateur d'équivalence (=).

Si la plupart des opérateurs maintiennent les résultats des opérations dans le même domaine, les trois opérateurs de comparaison <>= permettent de passer des domaines entier et de case vers le domaine des booléens. De ce fait, une opération telle que "$1 < 2$ *and* $nearby[1, 2] = dirt$", qui mélange les types de valeur, est tout-à-fait légale et renvoie un résultat booléen.

Ceci a pour conséquence qu'une expression peut mélanger des types et qu'il n'est pas possible d'en établir la valeur de validité avant d'être certain d'avoir traité tous les opérateurs : une lecture partielle pourrait en effet sembler renvoyer une expression invalide : "2 *and* $nearby[1, 2] = dirt$" est en effet inacceptable, puisqu'elle associe un entier à une comparaison booléenne.

Pour cette raison, nous avions introduit au point 2.3.2 une règle intermédiaire ostensiblement superflue : expression→subExpression. Cette règle nous permet de faire démarrer la visite des expressions par un point explicite, au retour duquel on peut calculer la valeur de validité. Si nous n'avions que la règle récursive, il nous serait impossible de savoir à l'exécution si la lecture de l'expression est terminée.

- Quand nous entrons dans le contexte de l'expression, nous pouvons démarrer la visite puis, au retour, vérifier si l'expression complète est valide.

be.unamur.info.b314.compiler.visitors.ExpressionVisitor#visitExpression

```
@Override
public Expression visitExpression(final GrammarParser.ExpressionContext ctx) {
    // Construire et verifier l'expression complete.
    return visitSubExpression(ctx.subExpression())
          .validate();
}
```

- Ensuite, nous entrons en récursion et nous traitons toutes les expressions sans nous soucier de leur validité partielle.

be.unamur.info.b314.compiler.visitors.ExpressionVisitor#visitSubExpression

```
@Override
public Expression visitSubExpression(final GrammarParser.SubExpressionContext ctx) {

    // Verifier si l'expression est une valeur simple.
    if (ctx.value() != null) {
        // Si oui, retourner la valeur.
        return visitValue(ctx.value());
    }

    // Sinon, recomposer l'operation.
    if (ctx.operation() != null) {
        // Recuperer l'operateur.
        final Operator operator = new OperatorVisitor().visitOperation(ctx.operation());
        // Recuperer l'operande gauche.
        final Expression left = visitSubExpression(ctx.subExpression(0));
        // Recuperer l'operande droite.
        final ExpressionLeaf right = (ExpressionLeaf)
             visitSubExpression(ctx.subExpression(1));
        // Construire et retourner l'operation.
        return left.append(right, operator);

    }
    throw new UnsupportedOperationException(ctx.getText());
}
```

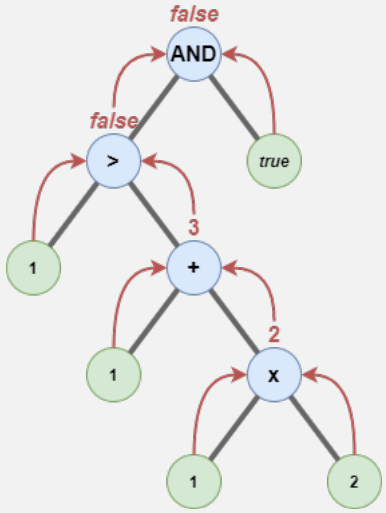
3.4.3 Calcul de la valeur

Une fois que l'arbre de l'expression a été reconstitué et sa validité affirmée, nous pouvons éventuellement en tirer la valeur. Pour ce faire, nous utilisons le mécanisme du pattern *Decorator*, qui consiste pour chaque expression à exiger la valeur de ses sous-expressions puis à calculer son propre résultat sur base de ses valeurs.

Nous avons ainsi les règles de récurrence suivantes :

- Une expression de type feuille renvoie sa simple valeur.
- Une expression de type noeud terminale (c'est-à-dire ayant deux feuilles comme extrémités) calcule simplement son résultat en effectuant l'opération mandatée par son opérateur sur les valeurs simples renvoyées par ses deux feuilles. Ce faisant, ce noeud se transforme en simple valeur et donc en feuille.
- Une expression de type noeud non-terminale ordonne à ses sous-expressions d'effectuer les calculs nécessaires et ainsi de se muer en feuilles dont elle va pouvoir utiliser la valeur.

On observe ainsi un mécanisme d'aspiration des résultats depuis les feuilles vers la racine.



3.5 Fonctions

3.5.1 Ajout dans le contexte

Comme mentionné dans le point 3.3.3, la déclaration des fonctions comprend également la déclaration des éventuelles variables locales, et des éventuels arguments. Nous retrouvons ici les notions d'environnement local et d'environnement global.

Par « environnement », nous entendons l'état courant de la table des symboles à un instant T de la lecture du programme, à savoir l'ensemble des déclarations qu'elle contient à cet instant du programme.

- L'environnement **global** correspond à l'ensemble des déclarations globales, c'est-à-dire les déclarations des variables globales et des fonctions.
- L'environnement local correspond quant à lui à l'ensemble des déclarations globales telles qu'elles se trouvaient dans la table des symboles avant l'entrée dans la fonction, mais également l'ensemble des déclarations locales de la fonction courante (arguments et variables locales).

Afin de passer d'un environnement à un autre, nous avons alors utilisé un système de stack.

Concrètement, l'entrée dans la règle « fctDecl » est détectée par le listener, et ce dernier fait alors appel à la fonction « enterFctDecl » présente dans le visitor.

Ensuite, avant de visiter une fonction, la table des symboles contenant les déclarations globales est sauvegardée dans une stack, de façon à sauvegarder l'environnement global tel qu'il était avant d'entrer dans la fonction.

Dans un second temps, une nouvelle table des symboles est créée, et celle-ci est initialisée avec le contenu de l'environnement global. Cette étape permet alors de réaliser deux choses :

- Créer un environnement local afin de stocker les différentes déclarations locales
- Stocker dans cet environnement toutes les déclarations globales obtenues précédemment, de façon à pouvoir y récupérer l'une ou l'autre information présente dans celui-ci. Ce cas de figure peut se présenter lorsque, par exemple, nous avons besoin de faire référence à une variable globale au sein de la fonction.

Lors de la visite d'une fonction, un objet de type Function est alors créé, et les attributs de celui-ci sont complétés au fur et à mesure de la visite de la fonction.

Lorsqu'une variable locale/un argument est déclaré(e), celui-ci est placé à la fois dans la table des symboles courante des variables, mais également comme attribut de la fonction courante.

Lors du placement d'une variable/d'un argument dans la table des symboles des variables, comme pour toutes les déclarations de variables, une vérification préalable est effectuée afin de s'assurer que le nom de cette variable n'existe pas encore. Or, étant donné que les règles du langage PILS autorisent le fait qu'une variable locale possède le même nom qu'une variable globale préexistante, nous vérifions également que ce potentiel conflit n'intervient pas au sein du même environnement.

Pour cela, nous utilisons un attribut appelé « level », présent dans la classe Variable, et qui nous permet facilement de déterminer si une variable est locale, ou si elle est globale. Cet attribut « level » est initialisé grâce à la taille de la stack. En d'autres termes, lorsque l'on se trouve dans un environnement local, la taille de la stack est de 1, puisque nous y avons stocké l'environnement global. L'attribut « level » de chaque variable locale/argument se voit donc attribuer la valeur « 1 » s'il s'agit d'une déclaration locale, et la valeur « 0 » lorsqu'il s'agit d'une déclaration globale. Ainsi, deux variables portant le même nom, mais n'ayant pas la même valeur pour l'attribut « level » ne sont donc pas considérées comme étant en conflit, puisqu'elles ne font pas partie du même environnement.

Également, afin de vérifier qu'une variable locale ne porte pas le même nom que la fonction dans laquelle elle est déclarée, l'attribut de l'objet Function courant se voit également, à ce stade-ci, attribuer la valeur « 1 ».

Enfin, lorsque la fonction a été entièrement lue, l'environnement local est alors détruit, l'environnement global est restauré, et la fonction est renvoyée dans le listener. Suite à cela, il ne reste alors plus, dans les différentes tables des symboles, de trace des déclarations locales de la fonction, ni de la fonction en elle-même. La dernière étape consiste ainsi à placer, dans la table des symboles globales des fonctions, la fonction renvoyée par le visitor.

Pour finir, on pourrait questionner la raison pour laquelle la fonction est placée dans la table des symboles locale lors de la visite de celle-ci. L'explication est que notre partie sémantique prévoit la possibilité d'instaurer un système d'appels récursifs. Cependant, cela implique qu'au moment de l'appel à une fonction, cette dernière doit exister dans la table des symboles courante. Par conséquent, cette fonction est placée dans la table des symboles locales avant même de récupérer les instructions de celle-ci, de façon à ce que, si l'une de ces instructions fait appel à la fonction, il soit déjà possible de retrouver sa déclaration dans la table des symboles courante. De plus, en cas d'appels récursifs, la différenciation des différents environnements successifs générés par la récursion est gérée par la variable « level », puisque que celle-ci nous indique le niveau de profondeur dans lequel nous nous trouvons.

3.6 Variable d'environnement

Le langage PILS repose pour son exécution sur une série de variables explicites, déclarées par le développeur, mais également sur un ensemble de variables implicites.

- Les variables de positionnement absolu (*longitude*, *latitude*, *taille de la carte*) et relatif (vis-à-vis de l'ennemi et du Graal),
- Les variables d'inventaire (compte des objets, niveau de vie),
- La variable de voisinage (*Nearby*).

Hormis l'arène, qui doit être déclarée explicitement dans un fichier .WLD, l'ensemble de ces variables ne peut jamais faire l'objet d'une déclaration : elles sont considérées comme systématiquement présentes et initialisées. Il est donc nécessaire, en PILS comme en NBC de générer ces

variables lors de la compilation, de sorte à ce qu'elles soient effectivement présentes.

Nous avons pour cela listé les variables d'environnement dans un objet et les avons associées au type de données qu'elles sont censées contenir.

be.unamur.info.b314.compiler.pils.keywords.Property

```
/**
 * La liste des variables d'environnement
 */
public enum Property implements Reserved {

    ARENA("arena", Type.SQUARE, Arrays.asList(1, 1)),
    MAP("map", Type.INTEGER, Collections.emptyList()),
    RADIO("radio", Type.INTEGER, Collections.emptyList()),
    AMMO("ammo", Type.INTEGER, Collections.emptyList()),
    FRUITS("fruits", Type.INTEGER, Collections.emptyList()),
    SODA("soda", Type.INTEGER, Collections.emptyList()),
    LIFE("life", Type.INTEGER, Collections.emptyList()),
    LATITUDE("latitude", Type.INTEGER, Collections.emptyList()),
    LONGITUDE("longitude", Type.INTEGER, Collections.emptyList()),
    GRIDSIZE("grid", Type.INTEGER, Collections.emptyList());

    private final String token;
    private final Type type;
    private final List<Integer> dimensions;

    Property(final String token, final Type type, final List<Integer> dimensions) {
        this.token = token;
        this.type = type;
        this.dimensions = dimensions;
    }
    ...
}
```

De cette manière, à l'ouverture du programme, nous sommes en mesure d'insérer automatiquement les variables d'environnement nécessaires en début de traitement de programme et celles-ci sont disponibles pour le traitement.

3.6.1 Une variable d'environnement nommée Arena

La variable *Arena* est un objet central dans le programme PILS : elle permet de situer le joueur dans un contexte, de lui permettre d'interagir avec celui-ci (par déplacement ou par action) et de représenter l'état du jeu.

Il s'agit d'une variable d'environnement, ce qui signifie que son nom est réservé et qu'un développeur / codeur de fichier .B314 ne peut pas l'instancier. La variable arène ne peut ainsi être déclarée et assignée qu'au sein d'un type de fichier spécifique (les fichiers de monde à l'extension, en .wld). Pour pouvoir utiliser dans une variable Arena dans une stratégie, un développeur PILS doit donc rédiger un fichier .wld puis l'importer au sein de son fichier .B314.

Présentation de l'objet Arena

La variable *Arena* répond à des critères de validité stricts :

- L'arène doit être une variable de type SQUARE.
- L'arène doit être une variable de type tableau, à deux dimensions.
- Les deux dimensions doivent être égales, de sorte à présenter un tableau carré.
- L'arène doit contenir un certain nombre d'éléments en nombre restreint ou minimum :
  - *Exactement 1* : PLAYER, ENEMY, GRAAL
  - *Minimum 1* : MAP, RADAR, ZOMBIE

Pour pouvoir contrôler et manipuler cette variable, nous avons construit un objet spécifique :

be.unamur.info.b314.compiler.arena.Arena

```
/**
 * Un objet representant la surface du plateau de jeu.
 *
 * @see Square
 */
public class Arena {
    /**
     * La variable d'environnement correspond a l'arene de jeu.
     */
    private Variable variable;

    /**
     * Verifie si l'arene courante est valide d'un point de vue semantique.
     *
     * @return l'arene courante si elle est valide.
     * @throws InvalidArenaException si l'arene ne repond pas a l'un des criteres de validite.
     */
    public Arena validate() {

        assertIsValid();

        assertContainsOnlyOne(Square.PLAYER);
        assertContainsOnlyOne(Square.ENEMY);
        assertContainsOnlyOne(Square.GRAAL);
        assertContainsAtLeastOne(Square.ZOMBIE);
        assertContainsAtLeastOne(Square.MAP);
        assertContainsAtLeastOne(Square.RADAR);

        return this;
    }
    ...
}
```

Cet objet *emballe* la variable d'environnement et lui confère des méthodes permettant d'interagir avec cette variable de manière spécifique : récupérer la position du joueur, obtenir la taille de la carte, trouver un adversaire, et surtout vérifier que la carte répond aux critères énoncés ci-avant.

Lorsque nous terminons la lecture du fichier .wld (c'est-à-dire que nous atteignons la règle *exitWorld*), nous récupérons la variable d'environnement (si elle existe) et testons sa validité.

be.unamur.info.b314.compiler.listeners.PilsToNbcConverter#exitWorld

```
@Override
public void exitWorld(final GrammarParser.WorldContext ctx) {
    // On excute le programme
    ((World) program).run();
    // On rcupre le plateau de jeu constitu par la dclaration du monde et on dtruit le
        contexte.
    context.setArena();
    context.getArena().validate();
    context.popAndRestore();
    context.canDeclareArena(true);
    context.put(context.getArena().getVariable());
    context.canDeclareArena(false);

    log.info("World processed.");
    super.exitWorld(ctx);
}
```

- Si la variable est nulle ou
- si la variable est invalide :

Nous renvoyons une exception et nous déclenchons une sortie abrupte du programme. Autrement, nous prenons cette variable et nous la propageons au contexte supérieur.

C'est de cette manière que nous pouvons transmettre l'arène au fichier de stratégie, puisque celui-ci peut déclencher, à travers un import, la lecture de la carte.

De la déclaration à l'exécution

Pour pouvoir évaluer la validité de la variable, un obstacle de taille se pose : le fichier .wld n'est pas une simple déclaration de la carte : il s'agit d'un *programme* dont l'exécution produit (entre autre) une variable *Arena*.

Pour pouvoir lire et récupérer l'arène, il faut donc *exécuter* ce programme : lire et interpréter les déclarations et instructions, assigner les valeurs aux variables et effectuer les opérations de calcul. Jusqu'à présent, nous avons simplement reconstruit le programme en mémoire, en établissant des liens entre les différents éléments et en vérifiant sa correction vis-à-vis de la syntaxe et de la sémantique. Il faut maintenant appliquer ces constructions et suivre le fil d'exécution jusqu'à la terminaison du programme.

Pour effectuer ceci, nous avons à nouveau suivi le principe de modularité : chaque composant du programme est responsable de sa propre exécution et délègue ce qu'il peut déléguer à ses composants subalternes. Notre point de départ est le monde.

be.unamur.info.b314.compiler.pils.program.World#run

```
public void run() {
    getStatements().forEach(Statement::run);
    getClauseDefault().run();
}
```

À cet endroit, nous avons implémenté une méthode *run*, dont la fonction est de faire exécuter chaque élément par délégation :

- Les *Statements* s'exécutent un par un jusqu'à la clause *Default*.
- Au sein des *Statements*, les instructions s'effectuent selon leurs règles propres :

  - Les instructions SET calculent et mettent à jour la valeur des variables,
  - Les instructions IF et WHILE choisissent des chemins d'exécution en fonction de leur garde et effectuent d'autres instructions en série,
  - Les appels de fonction déclenchent les corps des fonctions auxquelles ils font référence,
  - etc.
- Le contexte est alors mis à jour au fur et à mesure, tout comme le ferait un moteur d'exécution classique. En sortie de programme, nous avons alors un nouvel état, avec des nouvelles valeurs dans les variables.

  > Ré-exécuter le code n'est dès lors pas *idempotent* : puisque ses préconditions auront changé, il n'y aucune garantie que sa postcondition puisse demeurer inchangée.

Nous pouvons alors récupérer la variable Arena et tester ses conditions de validité.

Deuxième partie

Génération de Code

Chapitre 4

Présentation du langage NBC

NBC est le langage cible de notre compilateur. Nous avons procédé de la même manière que lors de l'analyse du langage source PILS. Nous avons construit le programme en mémoire dans le but de faciliter son utilisation dans le code. La structure est expliquée ci-dessous.

4.1 Structure

Constants

Le package *Constants* contient les classes permettant de créer des constantes de type numérique et de type String.

Keywords

Le package *Keywords* contient les mots-clés réservés au langage NBC. Ils sont représentés à l'aide d'énumérations.

Declarations

Le package *Declarations* contient les classes permettant de créer des macros et des segments.

Definitions

Le package *Definitions* contient les classes permettant de déclarer des variables, définir des structures, mais aussi des types. Ces déclarations et définitions doivent se trouver dans un segment.

Instructions

Le package *Instructions* contient les classes permettant de créer des *statements*. Ceux-ci peuvent être des instructions d'assignations, de comparaisons, etc.

Program

Le package *Program* contient la classe du même nom *Program* permettant de créer le programme NBC à l'aide de toutes les structures expliquées dans ces points.

Routines

Le package *Routines* contient les classes *Thread* et *Subroutine*. Celles-ci permettent de créer des subroutines et/ou des threads dans le programme.

Structures

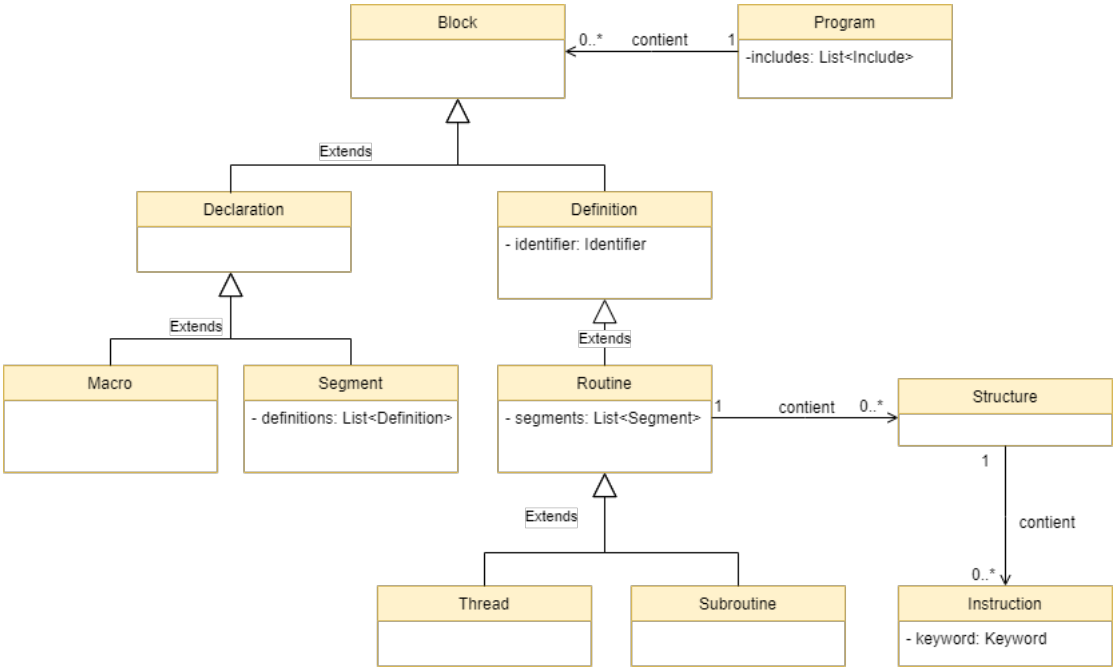
Le package *Structures* contient les classes permettant de construire des objets telles qu'une boucle *while*, une structure *when*, une condition *if* mais encore les structures gérant les différentes actions du robot.

Symbols

Le package *Symbols* contient l'énumération *Comparator* contenant des valeurs dans les instructions de comparaisons et d'autres. Il contient l'énumération *Operator* ainsi que les classes gérant les identifiants des threads, subroutines, etc.

4.2 Modélisation avec un diagramme de classe UML

Voici une représentation de la modélisation des classes du langage NBC que nous avons écrit. Celui-ci n'est pas complet et contient seulement les classes principales.



4.3 Conversion du langage PILS en NBC

Comme indiqué précédemment, afin de réaliser la traduction d'un code en langage PILS vers un code en langage NBC, nous avons commencé par construire le programme NBC en mémoire. Pour cela, nous avons utilisé différentes techniques.

4.3.1 Interface Function

L'architecture de notre compilateur est intimement liée au package *java.util.function*, et plus particulièrement à la classe *Function<T,R>*.

L'interface *Function<T,R>* est une interface générique dont le principe est d'accepter un certain type d'objet en entrée et de s'engager à produire un autre type d'objet en retour.

> *Represents a function that accepts one argument and produces a result.*
> *This is a functional interface whose functional method is apply(Object).*

Cet principe s'accorde assez bien avec la mécanique de compilation, puisqu'un compilateur s'engage en effet à accepter un objet d'un certain langage et à produire un objet dans dans un autre langage en retour.

4.3.2 Mappers

Les convertisseurs (*mappers*) sont les classes contenant toutes les méthodes responsables de la conversion des objets PILS vers des objets NBC correspondants, et sont des implémentations de l'interface Function.

On retrouve, parmi celles-ci, deux catégories de convertisseurs :

- Les convertisseurs **absolus**, qui n'ont pas besoin de faire appel au contexte pour générer un objet NBC.
- Les convertisseurs **relatifs**, ont besoin d'avoir accès au contexte, car ils nécessitent faire appel à d'autres convertisseurs pour construire leur objet NBC.

Afin de pouvoir réaliser leur travail de conversion, certains convertisseurs relatifs requièrent l'utilisation de convertisseurs spécialisés (les préparateurs, ou *preprocessors*). En effet, certaines instructions ne font pas l'objet d'une conversion de un à un. Il faut alors générer du code additionnel et des instructions intermédiaires.

Une première utilisation des préparateurs consiste à générer des instructions intermédiaires permettant de calculer la valeur finale d'une expression lorsque celle-ci est trop complexe pour calculée en une seule fois.

Prenons l'exemple de l'instruction set. En langage NBC, cette instruction commence par l'instruction « mov », suivie de deux termes que sont la variable à assigner (la destination), et la valeur à assigner (l'argument). Par conséquent, une opération comme « set test to x + 1 » ne peut être réalisée d'une seule traite en NBC, puisqu'elle nécessite la présence de trois termes.

L'ExpressionPreprocessor va alors devoir générer, au préalable, l'instruction calculant la valeur de « x+1 », avant de pouvoir l'assigner à la variable « test ».

Pour cela, il va tout d'abord commencer par aller rechercher, en mémoire, un registre libre afin de pouvoir stocker le résultat de l'opération « x+1 », et ensuite, retourner l'instruction permettant de stocker ce résultat dans ce registre.

La traduction en NBC donnera alors, dans ce cas-ci, le code suivant :

```
ADD integer1, x, 1
MOV test, integer1
```

Une autre utilisation du préparateur s'inscrit dans le cadre des appels de fonctions. En effet, si, dans le langage PILS, l'assignation des valeurs des paramètres formels aux paramètres effectifs se fait de façon transparente, il faut au contraire l'implémenter dans le langage NBC. Le FunctionCallPreprocessor va donc alors devoir se charger de générer les instructions nécessaires pour effectuer ces opérations.

Prenons l'exemple d'une fonction PILS dont la signature est la suivante :

```
functionTest1 as function (x as integer) : integer
```

et un appel à cette fonction :

```
set x to functionTest1(1)
```

En NBC, cet appel se traduirait alors par

- L'instuction assignant la valeur 1 à l'argument effectif « x »

  ```
  MOV functionTest1_x, 1
  ```

- L'instuction d'appel à la subroutine

  ```
  CALL functionTest1
  ```

- L'instruction assignant la valeur de retour à l'appelant

  ```
  MOV x, functionTest1_result
  ```

4.4 Des registres

Comme tout langage de programmation, les langages NBC et PILS fonctionnent avec des variables et des fonctions pour manipuler des données : en début de programme, des symboles sont déclarés et associent un nom à un type de données. Ensuite, un espace correspondant est réservé dans la mémoire avec une éventuelle valeur par défaut. Le programme peut alors utiliser ces symboles pour stocker, accéder à et transformer des valeurs au cours de l'exécution.

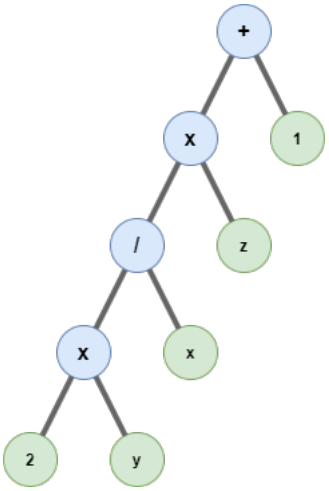
Ces deux langages sont dits fortement et statiquement typés : il n'est pas permis de définir un symbole sans en préciser le type et dès lors, il est possible de vérifier si les types sont respectés lors de toute étape d'assignation ou de calcul dès la phase de compilation.

Au-delà cette ressemblance, le langage NBC présente une caractéristique très *limitante* par rapport au langage PILS : alors qu'en PILS, un développeur peut assigner à des variables des valeurs issues d'expressions longues et complexes, en NBC, toute assignation n'accepte en argument qu'une seule entité : soit une expression mathématique constante à un ou plusieurs opérateurs, soit un et un seul élément symbolique (variable).

Dès lors, pour parvenir à traiter et assigner une expression complexe non-constante, il a fallu simplifier ces expressions en les décomposant en sous-expressions, en calculant et en stockant les résultats intermédiaires, puis en assemblant ceux-ci pour parvenir au résultat final.

Soit l'instruction PILS suivante :

```
set x to 2 * y / z + 1
```



Pour calculer son résultat en NBC, il faut procéder par itération, à la manière de l'aspiration présentée lors du traitement des expressions :

- Traiter les expressions les plus prioritaires en premier ($2 \times y$) et stocker leur résultat dans une variable temporaire ($int1$).

```
MUL int1, 2, y
```

- Remonter l'arbre en traitant successivement les opérations intermédiaires
  - $int1/x = int2$

```
SUB int2, int4, x
```

  - $int2 \times z = int3$

```
MUL int3, int2, z
```

  - $int3 + 1 = int4$

```
ADD int4, int3, 1
```

- Assigner le résultat final dans la variable.

```
MOV x, int4
```

> Pour pouvoir assigner ces valeurs dans des variables temporaires, il faut avoir préalablement déclaré ces variables. Or, il n'y a pas de déclaration explicite en PILS, puisque les expressions complexes sont supportées : il faut donc, en plus des variables normales, déclarer des variables temporaires suffisantes pour pouvoir traiter toutes les opérations requises par le programme PILS. Pour cela, nous devons calculer les besoins en espace-mémoire de ce programme.

4.4.1 Space Requirement

La problématique de réservation de l'espace-mémoire est inhérente à l'utilisation d'expression : il s'agit de déterminer le nombre de variables temporaires minimal pour la bonne tenue des opérations sans surcharger la mémoire de registres inutilisés.

Dans le cadre de l'opération d'assignation ci-dessus, nous avions réservé quatre variables pour la réalisation de l'opération. On remarque rapidement que les registres ne sont tous utilisés qu'une seule fois, alors que nous pourrions n'utiliser qu'un seul registre et accumuler le résultat au fur et à mesure des opérations : il y a gaspillage de ressources.
Il y a ainsi une tension constante entre d'une part la répartition de registres nécessaires lors de calculs simultanés et de résultats en attente d'affectation, et la réduction de la consommation par la réutilisation des registres rendus inutiles par la poursuite des opérations.

Pour parvenir à calculer correctement les besoins en ressources-mémoire, nous avons tout d'abord commencé par distinguer les besoins par type de données : une expression booléenne ne nécessitera en effet pas les mêmes types de registres qu'une expression entière.

Nous avons donc défini un besoin en termes d'espace comme l'objet suivant :

be.unamur.info.b314.compiler.mappers.SpaceRequirement

```
/**
 * Un prerequis-memoire indique le nombre de registres de chaque type necessaires a l'execution
      du programme courant dans le pire des cas d'utilisation.
 */
public class SpaceRequirement {

    /**
     * Un raccourci pour produire un prerequis vide.
     */
    public static SpaceRequirement NONE = new SpaceRequirement();

    /**
     * Un compteur du nombre de registres necessaires pouvant contenir des entiers.
     */
    private final int integer;
    /**
     * Un compteur du nombre de registres necessaires pouvant contenir des booleens.
     */
    private final int bool;
    /**
     * Un compteur du nombre de registres necessaires pouvant contenir des cases.
     */
    private final int square;

    /**
     * Un constructeur par defaut, qui initialise les besoins a zero registres dans toutes les
          categories.
     */
    public SpaceRequirement() {
        this.integer = 0;
        this.bool = 0;
        this.square = 0;
    }
    ...
}
```

Ensuite, pour déterminer les besoins propres à chacun des composants du programme PILS, nous avons choisi de procéder par division des responsabilités : chaque composant connaît ses besoins immédiats et délègue à ses composants internes le calcul de leurs besoins propres. Lorsqu'un composant contient plus d'un sous-élément, il faut réconcilier les besoins de chacun avant de remonter plus haut.

On distingue ici deux cas de figures :

- **Merge**
  Soit les sous-éléments se suivent en enchaînement, auquel cas il faut vérifier si les registres préalablement réservés sont suffisants (en nombre satisfaisant dans chaque type) et si non, ajouter les registres nécessaires selon un principe de *High Water Mark* : on compare les deux

prérequis et on sélectionne la valeur la plus haute dans chacun.

Par exemple, l'expression suivante nécessitera un registre entier et un registre booléen : l'expression entière du début nécessite un registre de type entier, même si celui-ci ne sera plus utilisé par la suite, et les trois opérations booléennes se font en ordre successif. On peut donc réutiliser le registre booléen sans se préoccuper de la perte de sa valeur précédente.

```
(1+x)<2 or y or z
```

- **Combine**
  Soit les deux éléments doivent être calculés simultanément et dans ce cas, il faut additionner les besoins en mémoire pour pouvoir accommoder les deux calculs en même temps.
  Par exemple, une instruction SET comprenant une variable NEARBY sous la forme suivante nécessitera deux registres de type entier pour stocker le résultat de chacun des indices, avant de pouvoir déterminer la valeur de case à assigner à y.

```
set y to nearby[x + 1, x * 2]
```

Une fois le besoin total du programme déterminé, nous pouvons construire la réservation en langage NBC en déclarant un segment avec autant de variable qu'il y a de registres de chaque type dans le prérequis.

be.unamur.info.b314.compiler.mappers.SpaceRequirement#getSpaceReservation

```
/**
 * Un producteur generant un registre NBC avec les definitions de registres necessaires pour
      accommoder les besoins indiques dans l'objet courant.
 *
 * @return Un segment avec des definitions de registres internes au fonctionnement du programme
      NBC.
 */
public Segment getSpaceReservation() {

    final Identifier integerTypeIdentifier = Type.SuperType.INT.getType().getIdentifier();
    final Stream<DefinitionVariable> integers = IntStream.range(1, integer + 1)
          .mapToObj(i -> new Identifier("integer" + i))
          .map(i -> DefinitionVariable.builder()
                .identifier(i)
                .typeName(integerTypeIdentifier)
                .build());

    final Identifier booleanTypeIdentifier = Type.SuperType.BOOL.getType().getIdentifier();
    final Stream<DefinitionVariable> booleans = IntStream.range(1, bool + 1)
          .mapToObj(i -> new Identifier("boolean" + i))
          .map(i -> DefinitionVariable.builder()
                .identifier(i)
                .typeName(booleanTypeIdentifier)
                .build());

    final Identifier squareTypeIdentifier = Type.SuperType.SQUARE.getType().getIdentifier();
    final Stream<DefinitionVariable> squares = IntStream.range(1, square + 1)
          .mapToObj(i -> new Identifier("square" + i))
          .map(i -> DefinitionVariable.builder()
                .identifier(i)
                .typeName(squareTypeIdentifier)
                .build());

    final List<DefinitionVariable> definitions = Stream.concat(integers,
         Stream.concat(booleans, squares))
          .collect(Collectors.toList());

    return Segment.builder()
          .definitions(definitions)
          .build();
}
```

On obtient alors un segment de la forme suivante :

```
dseg segment
       integer1 sdword
       boolean1 byte
       boolean2 byte
       boolean3 byte
dseg ends
```

4.4.2 Memory Manager

Les registres déclarés, il faut encore être capable de déterminer lesquels sont en cours d'utilisation et lesquels sont libres. Pour cela, nous avons mis en place un outil de contrôle : le *Memory Manager*.

be.unamur.info.b314.compiler.mappers.MemoryManager

```
/**
 * Le gestionnaire de memoire (Memory Manager) est l'instrument charge de controler l'attribution
      des registres de calcul d'expression lors de la conversion du programme PILS vers le langage
      NBC. Il rend compte du nombre de registres disponibles et de la quantite utilisee a tout
      moment dans la conversion du programme vers NBC.
 */
public class MemoryManager {
    /**
     * Le nombre de registres disponibles, par supertype.
     */
    private final Map<Type, Integer> availability = new HashMap<>();
    /**
     * Le nombre de registres deja utilise, par supertype.
     */
    private final Map<Type, AtomicInteger> reservation = new HashMap<>();
    ...
}
```

Le gestionnaire de mémoire tient à jour deux compteurs par type : un compteur du nombre de registres disponibles, ainsi qu'un compteur du nombre de registres utilisés.

Au début du traitement du programme NBC, on observe le prérequis mémoire et on "alloue" des emplacements dans le gestionnaire de mémoire.

be.unamur.info.b314.compiler.mappers.MemoryManager#allocate(SpaceRequirement)

```
/**
 * Associe de nouveaux registres a la memoire, sur base des besoins en espace donnes.
 *
 * @param requirement le compte des differents registres supplementaires necessaires.
 */
public void allocate(final SpaceRequirement requirement) {

    log.info("Allocating memory space...");

    final int integer = availability.get(Type.INTEGER) + requirement.getInteger();
    availability.put(Type.INTEGER, integer);

    final int bool = availability.get(Type.BOOLEAN) + requirement.getBool();
    availability.put(Type.BOOLEAN, bool);

    final int square = availability.get(Type.SQUARE) + requirement.getSquare();
    availability.put(Type.SQUARE, square);

    log.info("[{}]", availability);
}
```

Ensuite, durant la rédaction du programme NBC, les différents composants acquièrent et libèrent les registres au fur et à mesure de leur utilisation.

- **Acquisition** Un composant peut acquérir un registre s'il y en a au moins un de libre au moment où il effectue sa demande. À ce moment, le gestionnaire de mémoire réserve le registre, lui associe son nom unique et diminue le nombre de registres disponibles. Le composant peut alors utiliser le registre comme il l'entend.
- **Lecture** Un composant peut également demander à lire le dernier registre ayant été réservé, auquel cas le gestionnaire de mémoire lui retourne le nom du dernier registre du type demandé.
- **Libération** Enfin, une fois que le composant a terminé d'employer le registre, il retourne le contrôle au gestionnaire de mémoire qui le remet dans le groupe de registres disponibles.

4.5 Callable

Au sein du programme NBC, on retrouve différents éléments nominatifs : les routines (*threads*, *subroutines*) et les labels. Ces éléments forment des points d'ancrage (*Callable*) dans le programme NBC vers lesquels il est possible de détourner l'exécution.

Un simple programme comme ci-dessous possède deux chemins d'exécution : selon la valeur de la garde, le programme exécutera l'une ou l'autre assignation et évitera la seconde.

Simple IF instruction

```
BRTST elif, NEQ, guard
    MOV x, 1
    JMP end
elif:
    MOV x, 0
end:
```

> Ces éléments modifient la valeur du *Program Counter* : pour que l'assembleur puisse correctement calculer la valeur de destination, ils doivent dès lors être nécessairement unique. Ne pas respecter cette contrainte obligerait l'assembleur à choisir arbitrairement une valeur et pourrait donner lieu à des programmes non-déterministes.

Pour garantir l'unicité des noms utilisés au sein du programme, nous avons mis en place un gestionnaire de noms : la fabrique (*Name Factory*).

be.unamur.info.b314.compiler.mappers.NameFactory

```
/**
 * Le gestionnaire de noms est charge de suivre et de connaitre a tout moment la liste des noms
      deja utilises par le programme NBC et ne pouvant plus etre utilises par ailleurs.
 *
 * On distingue ici deux types de portee : les portees locales, internes a une routine et dont les
      valeurs ne doivent etre uniques qu'au sein de la meme routine, et les portees globales qui
      doivent etre respectees par l'ensemble des routines du programme indifferemment.
 */
public class NameFactory {

    /**
     * L'espace des noms a portee globale.
     *
     * @see Scope#GLOBAL
     */
    private final Namespace global = new Namespace();
    /**
     * L'espace des noms a portee locale.
     *
     * @see Scope#LOCAL
     */
    private Namespace local = new Namespace();
    ...
}
```

4.5.1 Namespaces

La fabrique tient à jour deux registres des noms utilisés :

- Un registre des noms à portée globale, qui doivent être uniques sur l'ensemble du programme (typiquement les identifiants de *thread*).
- Un registre des noms à portée locale, qui doivent être seulement uniques au sein d'une seule routine (Labels).

Ces registres sont appelés espaces nominatifs, ou *Namespaces*.

Le registre local est réinitialisé à chaque fois que le programme change de routine, tandis que le global reste d'application tout au long de la conversion.

4.5.2 Name Categories

Bien entendu, la fabrique n'invente pas des noms *ex nihilo* : des catégories de noms (*Name Categories*) sont définies et servent de base pour une génération déterministe des noms. La fabrique tient ainsi un compteur par catégorie de nom et génère des nouveaux couples {nom, indice} à chaque nouvelle production.
Tout comme le gestionnaire de mémoire, la fabrique de noms dispose de méthodes utilitaires pour fonctionner :

- **getCurrent**
  Une méthode pour obtenir le nom actuellement en cours d'utilisation, ce qui permet par exemple à une branche d'une clause WHEN de récupérer le même label de fin que toutes les autres, et d'éviter de multiplier les labels de sortie évitant la clause *Default*.
- **createNew**
  Une méthode pour créer et réserver un nouveau nom dans une catégorie donnée. La fabrique incrémente alors son compteur et retourne un nouvel identifiant unique.
- **resetLocal**
  Une méthode pour réinitialiser l'espace nominatif local, par exemple en sortie de *subroutine*.

Troisième partie

Conclusion

Chapitre 5

Réflexions sur le projet

5.1 Forces et faiblesses

5.1.1 Forces

Notre projet se distingue par sa forte modularité et par son degré de modélisation. Grâce à cette modélisation des langages PILS et NBC,

- Le code est plus explicite et structuré et il est plus facile d'appréhender les différents composants du programme grâce à la documentation qui y est disséminée.
- Il plus facile d'envisager la conversion vers le langage NBC puisqu'il est possible d'isoler chaque élément et l'associer avec une structure équivalente en NBC.
- La répartition du travail est plus simple, puisque chacun peut travailler sur un élément spécifique du code sans devoir maintenir l'ensemble de la chaîne.

Nous avons également veillé à nous appuyer sur des patterns de design (Function<T,R> et Walker) plutôt que dessiner et implémenter des mécaniques sur-mesure et peu transposables.

5.1.2 Faiblesses

Dans sa forme actuelle, notre développement présente plusieurs lacunes :

- Traitement des tableaux multidimensionnels
- Traitement de la récursion

Traitement des tableaux multidimensionnels

Notre programme ne permet pas de traiter des tableaux de données, que ce soit en déclaration/ initialisation, ou en lecture/écriture. Pour pouvoir traiter ce cas de figure, nous devrions introduire un traitement différencié lors de la conversion des déclarations.

- Déclaration/Initialisation : en plus de déclarer la variable "normale", une phase de pré-traitement au programme devrait avoir lieu pour traiter les variables-tableaux.
  1. Déterminer le nombre de tableaux unidimensionnels contenus dans la variable.
  2. Déclarer des variables internes pour chacune de ces lignes, pour lesquelles nous pourrions appeler l'instruction *arrInit* et initialiser la mémoire.
  3. Assigner les valeurs effectives au sein de ces variables (en utilisant *replace*).
  4. Regrouper ces variables au sein de la variable principale (en utilisant *arrBuild*).
- Lecture/Écriture : au lieu de simplement référencer la variable, il devrait y avoir comme pour les expressions une phase de pré-traitement pour récupérer le sous-tableau correspondant aux indices, puis accéder à la valeur (via l'instruction *index*) ou remplacer son contenu (via l'instruction *replace*).

Traitement de la récursion

Le langage PILS permet l'utilisation de fonctions récursives. Bien que cela ne soit pas interdit par le langage NBC, nous avons rencontré des difficultés à gérer une mémoire qui n'est pas allouable dynamiquement (du moins dans notre modèle) : toute variable doit avoir été explicitement déclarée dans le programme. Avec les fonctions récursives, il n'est pas possible de prédire le nombre d'exécutions récursives et de réserver explicitement un espace en conséquence. Notre code actuel réserve ainsi un espace fixe pour les fonctions et le moindre appel récursif entraîne l'écrasement des registres d'arguments.
Si la récursion intervient après que les registres aient perdus leur emploi, cela ne poserait pas de problème mais ce serait une limitation abusive.

5.2 Améliorations possibles

En termes d'amélioration, outre la poursuite des points précisés-ci avant, nous avons identifié trois pistes d'amélioration :

- Effectuer la vérification des contraintes du jeu avant de réaliser une action ("can move north" si le joueur est déjà en bout de tableau, "use map" ou "shoot north" si l'inventaire du joueur est vide).
- Effectuer la mise-à-jour des paramètres du jeu après l'action (modification de la longitude/ latitude, diminution de l'inventaire, etc.).
- Division de la classe *PilsToNbcConverter* en classes séparées pour gérer chaque partie du *GrammarBaseListener*. Exemple : *WorldListener*, *StrategyListener*, etc.

5.3 Apprentissage

Ce projet a été pour nous l'occasion de découvrir ANTLR4, un outil qui n'est habituellement pas mis en avant dans les milieux professionnels. Grâce à ce projet, nous avons pu mettre en pratique de nombreux éléments du cursus de programmation :

- Design d'une architecture et modélisation des éléments constituant le projet.
- Collaboration en synchronisation sur Git(Hub), communication (en direct et par documentation dans le code), et co-construction (*Peer Programming*).
- Découverte des design patterns du *Listener* et du *Visitor*.
- Mise en pratique du cours de syntaxe et sémantique, bien entendu, avec l'apprentissage du fonctionnement d'ANTLR et de la construction d'arbres syntaxiques sur base de cas concrets.

5.4 Commentaires

Ce projet n'a pas toujours été facile à gérer, sur le plan des tests : peu formés à l'utilisation de Jenkins, sans accès ou information sur la manière sont ils sont exécutés, limités dans nos possibilités de tests, nous n'avons pas toujours eu les moyens de réagir correctement aux situations remontées par l'outil d'intégration continue.

- Nous avons dû procéder à l'insertion de logs pour tenter de comprendre ce qui est exécuté et déterminer l'endroit provoquant l'erreur.
- Nous avons été obligés de pousser des changements temporaires sur la branche *master* pour tester une hypothèse (avant de faire un *reverse* et revenir à la situation "normale).

Ceci combiné à une disponibilité fluctuante du tunnel SSH, fonctionner avec une liste transparente et ouverte de tests nous permettrait de travailler plus efficacement : nous pourrions avoir une copie locale nous permettant de tester sur des branches de développement sans dépendre de l'infrastructure UNamur.

Annexes

Grammaire

Grammar.g4

```
grammar Grammar;

import Declarations
     , Imports
     , Instructions
     , Clauses
     , Words;

root: world | strategy;

world: DECLARE AND RETAIN
       (declaration)*
       instruction*
       clauseDefault;

strategy: DECLARE AND RETAIN
          impDecl?
          (declaration)*
          WHEN YOUR TURN
          clauseWhen*
          clauseDefault;

declaration: varDecl | fctDecl;
```

Actions.g4

```
lexer grammar Actions;

MOVE: 'move'|'MOVE';
SHOOT: 'shoot'|'SHOOT';
USE: 'use'|'USE';
NOTHING: 'nothing'|'NOTHING';
```

Clauses.g4

```
grammar Clauses;

import Expressions
     , Declarations
         , Instructions
         , Words;

clauseWhen: WHEN expression
            (DECLARE LOCAL varDecl+)?
            DO instruction+
            DONE;

clauseDefault: BY DEFAULT
               (DECLARE LOCAL (varDecl)+)?
               DO instruction+
               DONE;
```

Declarations.g4

```
grammar Declarations;

import Instructions
     , Types
     , Words;

varDecl: varType ';';

fctDecl: ID AS FUNCTION '(' (varType (',' varType)* )? ')' ':' (scalar | VOID)
        (DECLARE LOCAL varDecl+)?
        DO instruction*
           (RETURN (expression | VOID))?
        DONE;

varType: ID AS type;
```

Directions.g4

```
lexer grammar Directions;

NORTH: 'north'|'NORTH';
SOUTH: 'south'|'SOUTH';
EAST: 'east'|'EAST';
WEST: 'west'|'WEST';
```

Expressions.g4

```
grammar Expressions;

import Words;

reference: variable | function;

variable: ID ('[' expression (',' expression)? ']')?;

function: ID '(' (expression (',' expression)*)? ')';

expression: subExpression;

subExpression: subExpression operation subExpression | value;

operation: op=( '+' | '-' | '*' | '/' | '%' | AND | OR | '>' | '<' | '=' );

value: integer | bool | square | reference | '(' expression ')';

integer: neg='-'? NUMBER | intVariable;

intVariable : position | count | LIFE;

position: LATITUDE | LONGITUDE | GRID SIZE;

count: (MAP | RADIO | RADAR | AMMO | FRUITS | SODA) COUNT;

bool: boolValue | boolLocation | boolNegation;

boolValue: (TRUE | FALSE);

boolLocation: (ENNEMI | GRAAL) IS (NORTH | SOUTH | EAST | WEST);

boolNegation: NOT expression;

square: squareValue | squareNearby;

squareValue: DIRT | ROCK | VINES | ZOMBIE | PLAYER | ENNEMI | MAP | RADAR | RADIO | AMMO | FRUITS
     | SODA | GRAAL;

squareNearby: NEARBY '[' expression ',' expression ']';
```

Imports.g4

```
grammar Imports;

impDecl: keyword=('import'|'IMPORT') FILENAME;

FILENAME: LETTER (LETTER | DIGIT)*'.wld';

fragment LETTER: 'A'..'Z' | 'a'..'z' ;
fragment DIGIT: '0'..'9';
```

Instructions.g4

```
grammar Instructions;

import Expressions
     , Words;

instruction: skip='skip' # SkipInstr
           | IF expression THEN body (ELSE body)? DONE # IfInstr
           | WHILE expression DO body DONE # WhileInstr
           | SET variable TO expression # SetInstr
           | COMPUTE expression # ComputeInstr
           | NEXT action # NextInstr
;

body: instruction+;

action: (MOVE|SHOOT) (NORTH|SOUTH|EAST|WEST)
      | USE (MAP|RADAR|RADIO|FRUITS|SODA)
      | DO NOTHING;
```

Items.g4

```
lexer grammar Items;

MAP: 'map'|'MAP';
RADAR: 'radar'|'RADAR';
RADIO: 'radio'|'RADIO';
FRUITS: 'fruits'|'FRUITS';
SODA: 'soda'|'SODA';
AMMO: 'ammo'|'AMMO';
GRAAL: 'graal'|'GRAAL';

DIRT: 'dirt'|'DIRT';
ROCK: 'rock'|'ROCK';
VINES: 'vines'|'VINES';

ZOMBIE: 'zombie'|'ZOMBIE';
PLAYER: 'player'|'PLAYER';
ENNEMI: 'ennemi'|'ENNEMI';

LIFE: 'life'|'LIFE';

LATITUDE: 'latitude'|'LATITUDE';
LONGITUDE: 'longitude'|'LONGITUDE';
GRID: 'grid'|'GRID';
SIZE: 'size'|'SIZE';
COUNT: 'count'|'COUNT';
```

KeyWords.g4

```
lexer grammar KeyWords;

// Words

RETAIN: 'retain'|'RETAIN';
IMPORT: 'import'|'IMPORT';
BY: 'by'|'BY';
DEFAULT: 'default'|'DEFAULT';

WHILE: 'while'|'WHILE';
WHEN: 'when'|'WHEN';
DO: 'do'|'DO';
AS: 'as'|'AS';
SET: 'set'|'SET';
TO: 'to'|'TO';
COMPUTE: 'compute'|'COMPUTE';
NEXT: 'next'|'NEXT';
DONE: 'done'|'DONE';

IF: 'if'|'IF';
THEN: 'then'|'THEN';
ELSE: 'else'|'ELSE';
AND: 'and'|'AND';
OR: 'or'|'OR';
NOT: 'not'|'NOT';
TRUE: 'true'|'TRUE';
FALSE: 'false'|'FALSE';
IS: 'is'|'IS';

DECLARE: 'declare'|'DECLARE';
LOCAL: 'local'|'LOCAL';
GLOBAL: 'global'|'GLOBAL';
FUNCTION: 'function'|'FUNCTION';
RETURN: 'return'|'RETURN';

NEARBY: 'nearby'|'NEARBY';
COUNT: 'count'|'COUNT';
YOUR: 'your'|'YOUR';
TURN: 'turn'|'TURN';

// Comments -> ignored

COMMENT: '/*' .*? '*/' -> skip ;

// Whitespaces -> ignored

NEWLINE: '\r'? '\n' -> skip ;
WS: [ \t]+ -> skip ;
```

Types.g4

```
grammar Types;

import Words;

type: scalar | array;

scalar: INTEGER | BOOLEAN | SQUARE;

array: scalar '[' NUMBER (',' NUMBER)? ']';
```

Variables.g4

```
lexer grammar Variables;

// Fragments

fragment LETTER: 'A'..'Z' | 'a'..'z' ;
fragment DIGIT: ZERO | NONZERO;
fragment NONZERO: '1'..'9' ;
fragment ZERO: '0';

NUMBER: (DIGIT)+;

// Types

BOOLEAN: 'boolean'|'BOOLEAN';
INTEGER: 'integer'|'INTEGER';
SQUARE: 'square'|'SQUARE';
VOID: 'void'|'VOID';

// Common Variable Definitions

ID: LETTER (LETTER | DIGIT)*;
```

Words.g4

```
lexer grammar Words;

import Actions
     , Directions
     , Items
     , KeyWords
     , Variables;
```

Bibliographie

[1] *Java Operator Precedence*. URL : https://www.programiz.com/java-programming/operator-precedence#precedence-table.

[2] *NBC Programmer's Guide*. URL : http://bricxcc.sourceforge.net/nbc/doc/nbcapi/lang.html.

[3] Terence PARR. *ANTLR 4 Documentation*. URL : https://github.com/antlr/antlr4/blob/master/doc/getting-started.md.

[4] *Setting up Lombok with Eclipse and Intellij*. URL : https://www.baeldung.com/lombok-ide.