UNIVERSITÉ DE NAMUR

# Théorie des langages
# Syntaxe et sémantique
# Projet Compilateur

Groupe 13
CASTIAUX Julien
MINNE Denis
WARSZAWSKI Kenny

# Table des matières

Introduction....1
Démarche de travail....2
Architecture du compilateur....2
  Analyse lexicale, lexeur....2
  Analyse syntaxique, parseur....3
  Analyse sémantique, visiteur....6
    Table des symboles....6
    Parcours de l'arbre syntaxique....7
    Diagramme de classes....8
  Génération du code, traducteur....9
    Les actions *play+*....9
    Les variables....9
    L'évaluation des expressions mathématiques....10
    Les appels de fonction....10
    Résultat final du code....10
Structure du projet....11
Conclusion....12

## *Introduction*

Le projet de compilateur est un projet consistant à mettre en pratique les enseignements théoriques sur la nature des langages hors contextes et réguliers (cours de syntaxe et sémantique) ainsi que sur la structure des langages de programmation (cours d'analyse et de modélisation des systèmes d'information).

Ce projet vise à écrire un compilateur entre le langage *play+*, langage de haut-niveau, et le *NBC*, langage bas-niveau utilisé pour piloter les robots Lego Mindstorm.

Le langage *play+* est une forme évoluée du langage *play* utilisé au sein du jeu Cody développé par la société CodingSpark qui permet de déplacer un personnage "Cody" dans un environnement 2D pour que celui-ci progresse en évitant différents obstacles ou ennemis jusqu'à un trésor. Le langage est caractérisé par des actions comme `left`, `up`, etc qui permettent de déplacer le personnage mais également par des mots clés comme `if`, `repeat`, etc qui permettent de structurer le code. Le langage *play+* ajoute des variables, des types et toute une série d'éléments directement tirés de langages de programmation génériques comme le C.

## *Démarche de travail*

Notre démarche de travail a été de travailler ensemble en continu sur l'ensemble du projet. Nous avons principalement utilisé l'interface de Github au moyen des *issues*, *pull requests* et de la zone de commentaire sur les *commits* pour communiquer, proposer des changements et prévenir les potentielles erreurs. Une réelle force de notre équipe a été la communication asynchrone, au lieu de se fixer des rendez-vous pour travailler, chacun travaillait quand il avait du temps et passait en revue ce que les autres avaient fait. Nous nous sommes tout de même vus régulièrement à l'université pour planifier le travail et échanger sur ce qui avait été fait.

En matière de planning, nous nous sommes dans un premier temps concentrés sur la rédaction de la grammaire en nous basant à la fois sur notre projet du 1e quadrimestre et sur le document introductif communiqué par nos professeurs. Nous avons perdu beaucoup de temps à prendre en main l'outil *ANTLR* (décrit plus bas) ce qui nous a fait prendre du retard sur le projet. Une fois la grammaire relativement stable, nous nous sommes attaqués au visiteur tout en repassant régulièrement sur la grammaire pour faciliter le développement dudit visiteur. Ensuite, lorsque la table des symboles était relativement stable, nous nous sommes attaqués à la génération du code.

## *Architecture du compilateur*

Sur recommandation de nos professeurs et assistants, nous avons utilisé la bibliothèque Java *ANTLR* en vue de la réalisation de ce travail. La bibliothèque *ANTLR* se présente comme "un puissant générateur de parseur pour lire, analyser, exécuter ou traduire du texte structuré" et est prisée par des grands noms du monde informatique comme Guido van Rossum, BDFL de Python.

*ANTLR* est donc un outil servant à générer des parseurs de langage. Les différentes étapes sont expliquées ci-dessous.

## Analyse lexicale, lexeur

La première étape de tout parseur consiste à sortir tous les *mots* qui constituent un *document*. Cette étape travaille sur le flux de caractères du document original et groupe les lettres en mots. Chaque mot, ou *lexème*, est détecté au moyen d'une *expression régulière*. Une expression régulière (ou *regexp* de l'anglais *regular expression*) est une formule basée sur la concaténation, l'union et la répétition des lettres d'un alphabet donné. Elle permet de détecter des lexèmes dans une suite de caractères. Mis ensemble, un jeu de *regexp* définit un dictionnaire.

Dans le dictionnaire français, la suite de caractères constituée des lettres `u` et `n` forme le mot `un` qui est l'article indéfini singulier masculin. Ce mot est détectable au moyen de la *regexp* `un` qui est la concaténation des lettres `u` et `n`.

Dans notre dictionnaire, la suite de caractères constituée des lettres `"hello"` est un `STRING`. Les `STRING` sont détectables au moyen de la *regexp* `"(~[\\,\r\n])+"` expliquée ci-dessous :

1) Le caractère `"`, concaténé à...
2) Une quantité non nulle de caractères différents de l'antislash, du retour chariot et du retour à la ligne, concaténés à...
3) Le caractère `"`.

La bibliothèque *ANTLR* utilise un Automate Fini généré sur base du langage afin d'opérer à l'analyse lexicale. Si un mot est inconnu du langage, une erreur surviendra.

## Analyse syntaxique, parseur

L'étape de l'analyse syntaxique consiste à structurer un flux de lexèmes dans un arbre syntaxique. Cette étape convertit une suite de mots en phrases et regroupe les phrases en texte structuré. Chaque phrase est détectée au moyen d'une *règle BNF* (de l'anglais *Backus-Naur Form*), une notation technique permettant de décrire la syntaxe d'une phrase. Une *règle BNF* est constituée d'éléments *terminaux* (des lexèmes) et d'éléments *non-terminaux* (d'autres règles). Mis ensemble, un jeu de règles constitue une grammaire.

Dans la langue française, la phrase "Le lapin mange une carotte." est une phrase valide car elle est de la forme sujet-verbe-complément. Un jeu de règle valide en notation *NBF* pourrait être le suivant:

```
ARTICLE_DEFINI_SINGULER_MASCULIN: '[Ll]e';
ARTICLE_INDEFINI_SINGULIER_FEMININ: '[Uu]ne';
NOM_COMMUN_LAPIN_SINGULIER_MASCULIN: 'lapin';
NOM_COMMUN_CAROTTE_SINGULIER: 'carotte';
VERBE_MANGER_INDICATIF_PRESENT_SINGULIER_3: 'mange';
POINT: '.';

déterminant: ARTICLE_DEFINI_SINGULER_MASCULIN
           | ARTICLE_INDEFINI_SINGULIER_FEMININ
           ;
nom_commun: NOM_COMMUN_LAPIN_SINGULIER_MASCULIN
          | NOM_COMMUN_CAROTTE_SINGULIER
          ;
verbe_conjugué: VERGER_MANGER_INDICATIF_PRESENT_SINGULIER_3;

sujet: déterminant nom_commun;
verbe: verbe_conjugué
complément: déterminant nom_commun;

phrase: sujet verbe complément POINT;
```

Les premières lignes (en majuscule) sont les terminaux, l'ensemble des lexèmes accompagné par leur expression régulière. Expliqué ci-dessus, on retrouve ainsi la définition de tous les mots que constitue notre dictionnaire.

Les lignes suivantes (en minuscule) sont les non-terminaux, l'ensemble des règles syntaxiques qui regroupent les *terminaux* en *classe* (`déterminant`, `nom_commun`, `verbe_conjugué`), les *classes* en *fonction* (`sujet`, `verbe`, `complément`) et pour finir les *fonctions* en phrase.

Dans notre grammaire, nous avons la notion d'expressions mathématiques. Une expression mathématique est définie comme suit:

```
// Expressions
exprDroit: exprBool;
exprBool: exprBool AMP AMP exprBool     # andExpr
        | exprBool PIPE PIPE exprBool   # orExpr
        | EXCL exprBool                 # notExpr
        | exprEnt eqOp exprEnt          # eqExpr
        | exprEnt compOp exprEnt        # compExpr
        | exprEnt                       # exprEnt_
        ;
exprEnt: signOp exprEnt             # signExpr
       | exprEnt factOp exprEnt     # factExpr
       | exprEnt termOp exprEnt     # termExpr
       | LPAR exprDroit RPAR        # parExpr
       | atomExpr                   # atomExpr_
       ;
// Operators
eqOp: EQUAL EQUAL | EXCL EQUAL;
compOp: LT | LT EQUAL | GT | GT EQUAL;
termOp: PLUS | MINUS;
factOp: STAR | SLASH | PERCENT;
signOp: PLUS | MINUS;

// Atomic value
atomExpr: funcCall | exprGauche | entierNat | CHR | STRING | TRUE | FALSE ;
```

On retrouve ici quatre familles :

- Les expression booléennes: et logique, ou logique, non logique, test d'égalité et comparaisons.
- Les expressions entières: négation, addition, soustraction, multiplication, division et modulo.
- Les opérateurs utilisés dans ces opérations: !=, <, `+`, `*`, …
- Les valeurs atomiques: retour de fonction, accès mémoire, entier naturel, caractère, chaine de caractères, vrai et faux.

L'ordre avec lequel chaque règle a été déclaré a de l'importance, il assure la correcte *priorité des opérations*. Au sein d'une même règle, les lignes supérieures ont priorité sur les lignes inférieures.

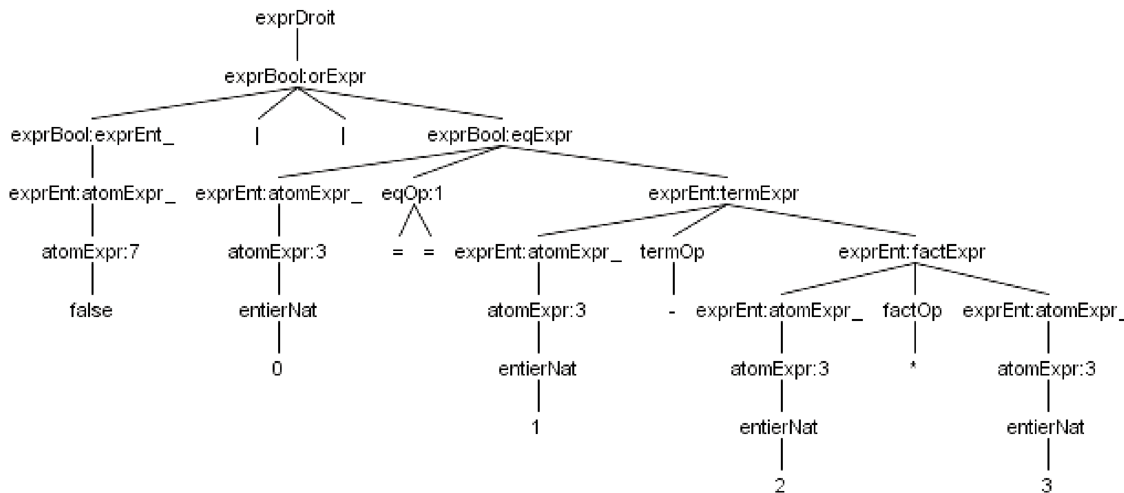
Ainsi, l'expression :

```
false || 0 == 1 - 2 * 3
```

s'évalue :

```
((false) || ((0) == (((1) - ((2) * (3))))))
```

La priorité des opérations est respectée.




*Arbre syntaxique de l'expression* `false || 0 == 1 - 2 * 3`

Au fil des règles, des classifications et des regroupements, l'analyse syntaxique crée un *arbre syntaxique*. Avec la règle primaire (`phrase` dans la phrase et `exprDroit` dans l'expression) comme tronc de l'arbre et les *terminaux* comme feuilles, l'ensemble des phrases se retrouve hiérarchisé. Le mot `lapin` est maintenant accessible via `phrase > sujet > nom_commun` tout comme `0` l'est via `exprDroit > exprBool#orExpr > exprBool#eqExpr > exprEnt#atomExpr_ > atomExpr`. C'est sur base de cet arbre que l'analyse sémantique de l'étape suivante pourra donner un sens au texte initial.

Comme le lexeur renverra une erreur pour un mot inconnu, une phrase ne correspondant à aucune règle n'est pas une phrase valide de notre langage et une erreur sera renvoyée.

## Analyse sémantique, visiteur

Le premier objectif de cette étape est de générer une table des symboles correspondant aux différentes déclarations. Ces déclarations peuvent être de différents types ainsi que de différentes visibilités. Tous ces éléments doivent transparaître dans la structure de la table des symboles.

Un autre objectif est de pouvoir vérifier que le code entré respecte certaines règles que la vérification syntaxique ne saurait vérifier automatiquement.

### *Table des symboles*

La structure de la table des symboles a été construite afin de laisser transparaître facilement la visibilité de l'ensemble des déclarations. La table des symboles est composée de 2 dictionnaires clé-valeur.

Un premier dictionnaire représente l'ensemble des déclarations globales. Le second reprend l'ensemble des procédures.

```
private Map<String, Declaration> globalDecl;
private Map<String, Declaration> functions;
```

*Table des symboles*

Chaque procédure contenue dans la collection `functions` va également contenir l'ensemble de ses déclarations. Ainsi, chaque procédure est capable de retrouver facilement ses déclarations locales.

```
public class Procedure extends Declaration {

    private Map<String, Variable> arguments;
    private Map<String, Declaration> declarations;
```

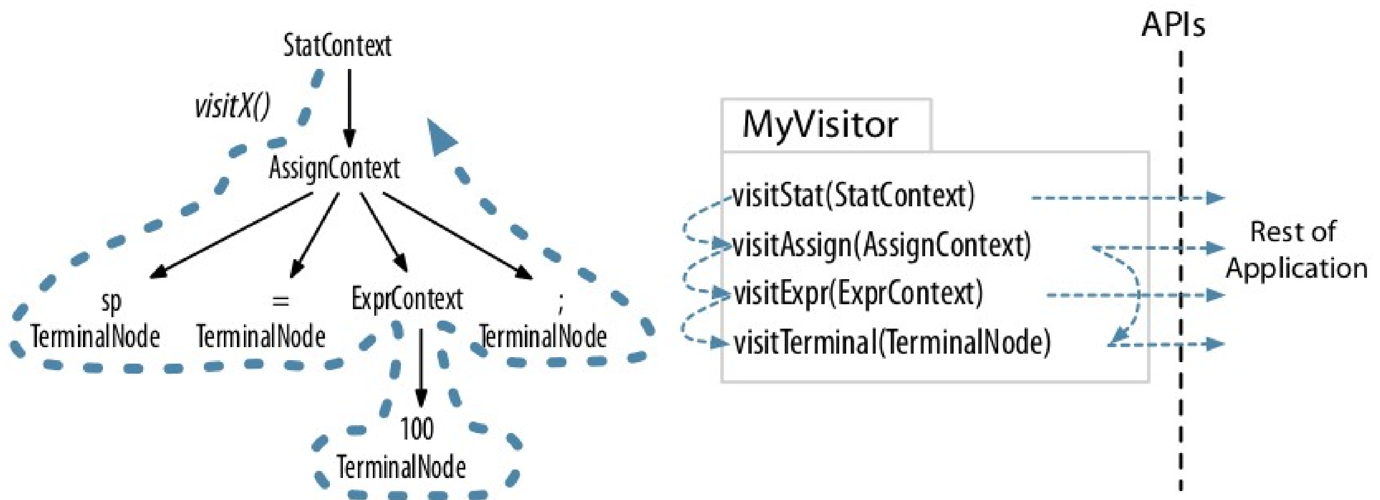
*Procédure*

### *Parcours de l'arbre syntaxique*

Afin de créer la table des symboles ainsi que de réaliser la validation des types, il est nécessaire d'appliquer une méthodologie pour parcourir l'arbre syntaxique généré précédemment. Certaines classes permettant de parcourir cet arbre sont générées automatiquement par l'outil *ANTLR*. Nous avons choisi d'utiliser les classes qui implémentent le pattern visiteur. Cette technique permet de mettre en application les notions théoriques vues lors du premier quadrimestre.

Cette méthodologie permet de mettre en lumière les notions d'attributs hérités et synthétisés. C'est-à-dire que certaines informations peuvent être issues d'un nœud père qui va donner à ses enfants l'ensemble de ses attributs. Parfois, les informations des enfants doivent être remontées au père. Dans ce cas, ces attributs doivent être synthétisés. Avec ces informations, il est tout à fait possible de créer la table des symboles ainsi que de valider les types en un seul passage.

L'arbre syntaxique va être parcouru de gauche à droite depuis sa racine de manière descendante en dérivation gauche. Une fois qu'un terminal est atteint, ses attributs sont synthétisés au nœud père et ensuite le nœud suivant est visité.




*Visiteur*

### *Diagramme de classes*

Afin de construire la table des symboles, une série de classes ont été implémentées pour représenter au mieux le langage *play+*. Ci-dessous, un diagramme de classes représentant les concepts du langage.

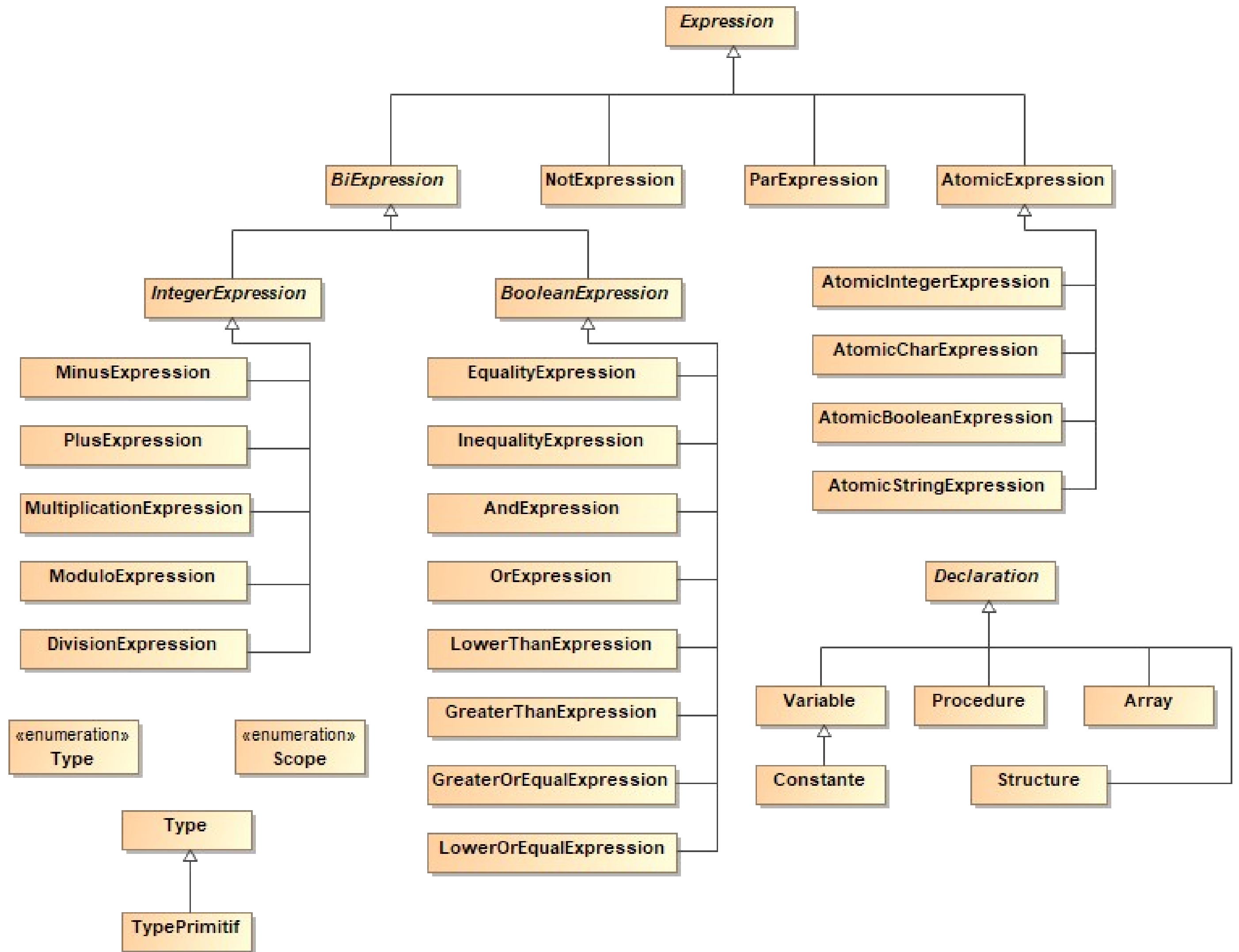



*Diagramme de classe*

## Génération du code, traducteur

Sur base de la carte du jeu, de l'arbre syntaxique et de la table des symboles, le traducteur consiste à générer un fichier d'instructions au format *NBC* dont la sémantique est équivalente à la sémantique du code source *play+*. Le fichier de sortie peut être chargé au sein d'un robot Lego Mindstorm (qui représente le personnage Cody) et évoluer dans un environnement réel au fil des instructions.

Plusieurs incompatibilités ont freiné la traduction. Parmi celles-ci on retrouve les actions *play+*, la portée des variables, le moteur mathématique et les appels de fonction.

### *Les actions play+*

Les actions *play+* `left`, `right`, `up`, `down`, `jump`, `dig` et `fight` n'existent pas dans *NBC*. Il a été nécessaire d'implémenter des routines dédiées qui sont injectées dans le code de sortie.

Les routines peuvent :

- effectuer un déplacement,
- se tourner dans la direction voulue,
- vérifier la nouvelle position sur la carte et agir en conséquence.

Pour se tourner à droite, la routine `cody_right` peut être appelée. Cette routine :

1) tourne le robot à droite en actionnant les moteurs,
2) avance le robot d'une case en actionnant les moteurs,
3) vérifie que le robot est toujours dans les limites de la carte,
4) vérifie que le robot est sur une position valide de la carte.

Si le robot n'est plus sur une position valide (eau, buisson, arbre, ...) ou hors de la carte, le robot se met à tourner dans le sens inverse des aiguilles d'une montre pour indiquer l'échec.

Si la commande `dig` est exécutée sur le trésor, le robot se met à tourner dans le sens des aiguilles d'une montre pour indiquer le succès du programme.

Pour éviter que les routines injectées n'entrent en conflit avec les fonctions *play+* du même nom, toutes les fonctions sont préfixées par `APP_` : la fonction *play+* `main` devient ainsi la fonction *NBC* `APP_main`.

### *Les variables*

Toutes les variables en *NBC* ont systématiquement une portée globale. Ceci entraine le même conflit que cité plus haut relatif au nommage desdites variables. Des variables appartenant à des fonctions différentes ne doivent pas pointer le même emplacement mémoire.

Pour empêcher cela, les variables globales sont préfixées par `APP_GLOBAL_` et les variables locales par `APP_LOCAL_<fct_name>_`. Il est donc impossible qu'elles puissent entrer en conflit entre elles ou avec les variables du système.

Il reste cependant le problème d'appels de fonction récursifs pour lequel aucune solution n'a été trouvée par manque de temps.

### *L'évaluation des expressions mathématiques*

Les instructions du *NBC* ne permettent qu'un calcul par instruction et ces calculs ne sont qu'à deux opérandes. Si on écrit `y = 3 * x + 5`, cette opération doit se faire en 3 parties :

1) multiplier 3 par `x` et sauvegarder le résultat temporairement,
2) additionner le précédent résultat temporaire et 5 et sauvegarder le résultat temporairement,
3) sauvegarder le résultat temporaire précédent dans `y`.

En *NBC*, cela donne les instructions suivantes :

```
mul TMP_INT_VAR0, 3, APP_main_x
add TMP_INT_VAR1, TMP_INT_VAR0, 5
mov APP_main_y, TMP_INT_VAR1
```

Les variables temporaires permettent d'effectuer toutes les opérations étape par étape.

### *Les appels de fonction*

*NBC* ne permet que d'exécuter des procédures qui ne contiennent pas de paramètres. La transmission de paramètres se fait uniquement via les variables qui sont toutes globales.

### *Résultat final du code*

Malheureusement, par manque de temps, tout ne fonctionne pas. Dans le code actuel, les variables temporaires ne peuvent être utilisées (et réservées) que pour l'instruction *play+* qu'on compile. Toutes sont réutilisables à l'instruction suivante. Il aurait fallu pouvoir réserver les variables pour des durées plus longues. Dans l'évaluation de l'expression `y = 3 * mafct() + 7`, il y aura un conflit entre les réservations de variables réservées pour cette instruction et les réservations de variables temporaires dans `mafct()`. Les retours des fonctions ne fonctionnent également pas. De plus, il aurait fallu calculer le nombre de variables temporaires nécessaires pour ne réserver que ce qu'il fallait.

## *Structure du projet*

Afin de mettre en pratique la partie sémantique et génération de code, il a fallu définir et implémenter des classes Java qui se trouvent au sein du package `be.unamur.info.b314.compiler`.

La partie sémantique se compose de deux classes principales :

- `DefaultPlayPlusVisitor.java` : Visite la grammaire à traiter, en vérifie les règles sémantiques et construit la table des symboles.
- `SymTable.java` : Table des symboles.

Toutes les classes liées à la construction de la table des symboles se retrouve dans le package `be.unamur.info.b314.compiler.language`

La partie génération de code se compose des classes :

- `NBCVisitor.java` : Visite la grammaire à traiter et utilise NBCPrinter afin d'écrire le code NBC final.
- `NBCVariable.java` : Ecrit des codes de gestion des variables par NBC (tableaux, structures, ...).
- `NBCOperation.java` : Décompose les opérations complexes en plusieurs calculs simples en utilisant des variables temporaires.
- `NBCPrinter.java` : Écriture du code NBC Final.

## *Conclusion*

Par manque de temps, nous n'avons pas mené le projet à son terme. La partie génération de code n'est pas capable de fidèlement sortir un programme sémantiquement identique au code source. Comme expliqué ci-dessus, le moteur de calcul ne fonctionne que sur des expressions simples et il est impossible d'avoir des fonctions récursives. L'analyse sémantique n'est pas totalement aboutie non plus, les énumérations et les types utilisateurs ne sont pas gérés. Pour finir, la grammaire pourrait être simplifiée si on relâchait certains tests unitaires.

Concernant le moteur de calcul, une solution envisagée est de convertir les expressions *infix* en *postfix* au niveau du visiteur au moyen de l'algorithme Shuting-yard. La notation *postfix* a l'avantage d'être efficacement calculable par un ordinateur et ne nécessite qu'une seule pile.

Concernant les fonctions, une implémentation différente basée sur une pile aurait pu être envisagée. Chaque appel de fonction aurait simplement empilé un étage pour mettre en mémoire les variables locales et le retour de la fonction. Étant donné que la structure de pile est absente de NBC, nous aurions dû l'implémenter nous-mêmes sur des tableaux.

Les énumérations et les types utilisateurs n'ont pas été implémentées du fait du manque de spécification à leur égard. Vu qu'aucun d'entre nous ne développe en C/C++, nous ne sommes pas à l'aise quant à la déclaration et l'utilisation de ces deux déclarations. Si ces deux déclarations sont tout de même présentes au sein de la grammaire, elles ne sont pas comprises par le visiteur.

Pour finir, nous ne sommes pas entièrement satisfaits de notre grammaire. À cause de certains tests unitaires envers lesquels nous sommes critiques, la grammaire contient des tournures de règles qui contrastent avec le reste. Deux des tests que nous trouvons invalides sont l'instruction `dig()` et les types au sein des expressions. Ces tests ont d'ailleurs complexifié grandement notre grammaire et nous avons perdu un temps bête à essayer de les passer. Nous avons au final pris la décision d'opérer aux vérifications au niveau de l'analyse sémantique.

Pour conclure, nous avons énormément appris au fil de ce projet. La science des compilateurs est une science nouvelle pour nous trois et malgré le nombre d’heures passée à ce projet, nous n'en n'avons fait qu'effleurer la surface. Comme tout projet, celui-ci a également mis l'accent sur la collaboration et l'échange des idées.