Théorie des Langages : Syntaxe et Sémantique

Spécification du Langage: *PILS*

A Simple Language for Imperative Programming

Créateur : James Ortiz

Les développeurs savent pourquoi!

"Jamais la programmation n'a été aussi populaire"

Faculté d'informatique
Université de Namur

Année académique 2020 - 2021

Table des matières

---

1 Présentation

Le cours de "Théorie des Langages : Syntaxe et Sémantique" a pour objectif principal de vous familiariser avec les concepts nécessaires à la création, à l'utilisation et à l'étude des langages de programmation. Le cours vous propose à la fois et simultanément une approche théorique (avec le cours et les séances de travaux pratiques) et une approche pratique (avec le projet compilateur, qui vous est présenté dans ce document).

L'objectif du projet compilateur est de coder un *compilateur*. Un compilateur est un *traducteur* d'un langage vers un autre. Il traduit tout programme écrit dans son langage source vers un programme équivalent dans son langage cible.

Dans ce projet, nous vous demanderons de coder un compilateur d'un type de langage source un peu particulier appelé *DSL* (pour *Domain Specific Language*) vers un langage machine (Next Byte Codes (NBC)). Un DSL est un *langage de programmation à l'expressivité limitée et se concentrant sur un domaine particulier* [2]. Par exemple, le HTML est un DSL utilisé pour décrire des documents hypertextes qui sera *interprété* par un navigateur ; un autre exemple est le langage utilisé par ANTLR [6] pour décrire une grammaire et qui lui est *compilé* par celui-ci en un semble de fichiers Java permettant de traiter un texte respectant ladite grammaire.

Pour cela, nous vous fournissons (sur WebCampus) :

— Un document (intitulé *Spécification du langage : PILS "Jamais la programmation n'a été aussi populaire"*) spécifiant de manière formelle la syntaxe et la sémantique du DSL d'entrée de votre compilateur.

— Un document (intitulé *NeXT Byte Codes Manual*) décrivant l'utilisation des instructions disponibles.

Afin de coder votre compilateur, nous vous imposons une série de *contraintes techniques et organisationnelles*. Ces contraintes, ainsi que les modalités d'*évaluation* de votre travail, sont détaillées dans la suite de ce document.

2 Contraintes techniques

Nous vous demandons de coder en Java un compilateur qui procède en 3 étapes successives :

1. Analyse lexico-syntaxique (avec création d'un arbre syntaxique abstrait) (20%).
2. Analyse sémantique (avec création d'une table des symboles) (30%).
3. Génération de code cible (sans optimisations) (50%).

La troisième étape ne doit être effectuée que si le code reçu en entrée par le compilateur est syntaxiquement correct.

2.1 Analyse lexico-syntaxique

Vous *générerez* un *analyseur lexico-syntaxique* à l'aide d'*ANTLR* [6], que vous utiliserez pour créer l'arbre syntaxique abstrait. Il s'agit d'écrire et d'utiliser une spécification (*i.e.*, une grammaire définie dans un fichier `.g`) qui :
— reconnaît tous les mots clés du langage (ponctuations, mots clés, identificateurs, etc.),
— ainsi que la structure grammaticale du code en entrée,
— tout en créant l'arbre syntaxique abstrait correspondant.

2.2 Analyse sémantique

Nous vous demandons de réaliser toutes les *vérifications de correction syntaxique non réalisées lors de l'analyse lexico-syntatxique* par parcours de l'arbre syntaxique abstrait construit par cette analyse lexico-syntatxique. Pour cela, nous vous demandons de construire et d'utiliser une *table des symboles* rassemblant des informations sur les identificateurs (noms de variables, de fonctions, ...). Vous choisirez intelligemment quelles informations vous stockerez dans cette table et sous quelle forme vous le ferez [5, 6].

2.3 Génération de code

Votre compilateur n'effectuera la phase de génération de code que *si le code reçu en entrée est syntaxiquement correct*. Étant donné un code source à l'entrée standard, votre compilateur fournira "OK" suivi d'un retour à la ligne sur l'erreur standard si et seulement si le programme est conforme à la spécification. Si le programme est non conforme à la spécification, votre compilateur fournira le message "KO" suivis d'un retour à la ligne sur l'erreur standard.

Dans le cas où votre compilateur a accepté le programme en entrée et imprimé "OK" sur l'erreur standard, vous devez ensuite *générer le NBC code* correspondant dans un fichier dont le nom (path) *est donné*.

Remarques.
— L'écriture de "OK" sur l'erreur standard se réalise à l'aide de l'instruction suivante :

```
System.err.println("OK");
```

— Si le programme n'est pas correct, vous pouvez faire précéder le "KO" sur l'erreur standard par tous les messages d'erreurs de votre choix. Par exemple :
```
Error : Variable 'charlie' not found
KO
```

3 Contraintes organisationnelles et évaluation

Le projet est réalisé *par groupe de 3 personnes* et l'évaluation porte sur les critères suivants :

1. Votre respect des exigences fonctionnelles, dans les délais imposés.
2. Votre respect des exigences non-fonctionnelles.
3. La remise d'un rapport écrit complet, correct et soigné dans le délai imparti.

4. La réussite d'une interrogation orale individuelle, portant sur l'ensemble du travail.

La note est individuelle (c'est-à-dire que des étudiants appartenant au même groupe peuvent obtenir des notes différentes). *Attention* : il est demandé à tous les membres d'un groupe un minimum de participation pour le codage et le test du compilateur (la seule réalisation du rapport ne constituera *pas* une participation suffisante pour une personne en particulier).

3.1 Exigences fonctionnelles

Les exigences fonctionnelles pour ce projet peuvent être résumées ainsi : construire un compilateur DSL vers PCode, conforme aux spécifications qui vous ont été fournies pour ces deux langages, et qui respecte les contraintes techniques données précédemment.

L'évaluation de ces contraintes fonctionnelles se fera en deux parties :

— la première teste que votre compilateur vérifie correctement la syntaxe du code source qui lui est fourni en entrée (message OK/KO sur l'erreur standard) ;
— la seconde teste que le code produit par votre compilateur est correct.

Sous-séries de tests. Afin de tester les différentes fonctionnalités du langage, nous avons divisé les tests fonctionnels en sous-séries. Chacune de ces sous-séries comporte des tests spécifiques à la *syntaxe* (vérification du code source en entrée et impression du message OK/KO) et à la *sémantique* (test du code produit par le compilateur). Les sous séries, organisées par ordre de dépendance (*i.e.*, la sous série n peut utiliser des instructions testées dans la sous série $n-1$) :

1. le programme minimal (programme vide avec uniquement la clause **by default**), l'instruction vide (**skip**) et les commentaires dans le code,
2. la déclaration de l'instruction (**import** FileDecl),
3. la déclaration de variables globales, en ce compris les tableaux (**declare and retain**),
4. les expressions droites (instruction **compute** ExprD) entières, booléennes et de type case (en ce compris les tableaux),
5. l'affectation (instruction **set** ExprG **to** ExprD) ;
6. la spécification du prochain coup à jouer (instruction **next** Action),
7. l'instruction conditionnelle (**if** ExprD **then** Instruction $^+$ **done** et **if** ExprD **then** Instruction $^+$ **else** Instruction $^+$ **done**),
8. la boucle (**while** ExprD **do** Instruction $^+$ **done**),
9. la déclaration de fonctions,
10. la spécification de clauses **when** (**when** ExprD),
11. la déclaration de variables locales aux fonctions et aux clauses **when** (**declare local**),

3.2 Exigences non-fonctionnelles

Plusieurs critères "de bonne conduite" seront évalués :

— *Qualité du code*, notamment :
  — Spécification (en utilisant le standard Javadoc ou celui vu au cours de Conception et Programmation Orientée Objet pour les commentaires dans le code) et abstraction [1] (montrant, par exemple, que vous avez réfléchi à la structure de l'implémentation au lieu de foncer tête baissée,...).
  — Propreté du code, par exemple : noms de variable significatifs, conventions de nommage cohérentes, respect des indentations, ... (à noter que ceci est facile à mettre en place avec un IDE moderne).

1. Un diagramme de classes à un moment ou l'autre est donc un plus.

  - la couverture de vos tests : 80% des méthodes, 70% des lignes et 60% des conditions couvertes avec Cobertura (la commande `mvn cobertura:cobertura` génère le rapport de couverture dans le dossier `target/site/cobertura` du projet Maven)
  - Accessibilité pour l'utilisateur, (par exemple : messages d'erreur clairs,...).
- Le respect des bonnes pratiques de l'orienté objet, en particulier en ce qui concerne les principes étudiés au cours de Conception et Programmation Orientée Objet.
- Le respect des présentes consignes pour le projet.

3.3 Rapport écrit

Le rapport sera soumis en version électronique (pas de version papier !), via WebCampus (rubrique travaux). Il sera rédigé dans un français clair, correct et précis. Un soin particulier sera apporté à ce que le rapport soit pratique et agréable à lire. Le but ici est de donner une synthèse du travail réalisé, soyez donc complet, mais concis. Le rapport contiendra les éléments suivants :

1. sur la 1^{er} page, Le titre, votre nom et prénom ainsi qu'une introduction.
2. Une description de la *démarche générale* que vous avez adoptée pour construire votre compilateur, ainsi que les choix faits pour l'implémentation. On vous demande ici de jeter un regard critique sur la manière dont le projet s'est déroulé au sein de *votre* groupe : quelle démarche avez-vous adoptée ? comment vous êtes-vous organisés pratiquement ? était-ce, à posteriori, une bonne idée ? etc.
3. Une description de la structure de donnée et de l'utilisation de votre table des symboles avec la/les définition(s) du/des classe(s) utilisée(s) pour l'implémenter et une justification de votre choix. Inutile de reprendre ici l'ensemble de la documentation, le but est d'expliquer *comment se structure la table des symboles et comment elle a été utilisée* pour valider le code en entrée du compilateur (pensez donc à utiliser un formalisme adéquat, tel qu'un diagramme de classes).
4. Une description générale de l'architecture de votre compilateur (i.e., la découpe en *packages* et une courte description de ceux-ci), ainsi qu'une liste des *principales* classes avec pour chacune *1-3 ligne(s)* expliquant ce qu'elle représente et à quoi elle sert.
5. Une *conclusion* qui synthétise :
   - Les *forces et les faiblesses* du compilateur que vous avez construit.
   - Les *améliorations* que vous pourriez/devriez/aimeriez apporter à ce compilateur.
   - Ce que ce projet vous a *appris*.
   - Tout *commentaire* constructif sur le projet en lui-même qui vous semble pertinent.

3.4 Échéances

Nous vous demandons de construire un compilateur en trois étapes (pour la partie fonctionnelle) et en respectant les échéances suivantes :

- **7 mars 2021 :** remise d'une version intermédiaire du compilateur avec ses tests unitaires passant les sous-séries suivantes :
  - Syntaxe : 1, 2, 3, 4, 5 et 6.
  - Sémantique : /
- **11 avril 2021 :** remise d'une version intermédiaire du compilateur avec ses tests unitaires passant les sous-séries suivantes :
  - Syntaxe : 1, 2, 3, 4, 5, 6, 7, 8, 9, 11.
  - Sémantique : 1, 2, 3, 4, 5, 6, 7, 8, 9, 11.
- **11 mai 2021 :** remise du code source avec ses tests unitaires passant toutes les sous-séries de tests syntaxiques, sémantiques et la generation de code (via GitHub).

— **13 mai 2021 :** remise du rapport et de la version final du code source documenté, avec ses tests unitaires, passant toutes les sous-séries de tests syntaxiques, sémantiques et la generation de code (via WebCampus (rubrique Projet)).

Remarque. La réalisation d'un compilateur est un travail de longue haleine qui demande un investissement conséquent : ne tardez donc pas à démarrer vos travaux et soyez assidu et persévérant dans votre travail.

3.5 Interrogation orale individuelle

L'interrogation orale aura lieu durant la session de juin. L'interrogation est individuelle (chacun répond à son tour), mais vous passez par groupe (tous les membres de groupes sont présents au même moment). Chaque étudiant doit être capable de :

— Montrer sa bonne compréhension des objectifs et de l'énoncé du projet.
— D'expliquer l'entièreté de sa démarche et de son travail.
— D'expliquer le fonctionnement de son compilateur, tant en terme de fonctionnement général que des détails d'implémentation.
— De détailler le comportement de son compilateur sur un code exemple. En particulier, l'étudiant saura construire l'arbre syntaxique, la table des symboles et le code généré correspondant à ce programme.

3.6 Version Initiale et Jenkins

La version initiale (version 0) de projet est sur le lien suivant : `https://github.com/UNamurCSFaculty/2021_SyntaxeSemantique_Students`. L'outil logiciel d'intégration continu (Jenkins) est sur le lien suivant : `http://compilateur.info.fundp.ac.be:8080/`.

3.7 Outils de développement à utiliser

Git (`https://git-scm.com`). Afin de faciliter vos développements, un repostitory Git [1] sera fourni à chaque groupe. Il vous est demandé de ne pas utiliser d'autre système de gestion de sources et ce afin d'éviter toute fuite de code. En cas de plagiat, les **deux** groupes seront sanctionnés (!). Toute utilisation de code non développé par votre groupe vaudra 0 pour l'intégralité du projet concerné. Tout code commun à plusieurs projets vaudra 0 pour l'intégralité des projets concernés.

Maven (`https://maven.apache.org`). Maven est un outil permettant de gérer le *build* et le déploiement d'applications Java [3, 4]. Il repose sur le principe suivant : *conventions plutôt que configuration*, les systèmes, APIs, outils, etc. devraient avoir une comportement par défaut raisonnable, ils doivent juste fonctionner sans qu'il soit nécessaire de les configurer [3].

Afin de faciliter votre travail, nous vous fournissons un projet Maven *à utiliser comme base*. Ce projet est configuré pour produire un fichier `jar` exécutable et contient entre autres une classe `Main` qu'il vous faudra modifier ou compléter pour que votre compilateur soit complet [2]. Lors de l'appel au compilateur, il sera nécessaire de fournir un certain nombre d'arguments (la commande `java -jar mycompiler.jar -help` vous donnera la liste de ceux-ci, ainsi que leur description), *nous vous demandons de ne pas modifier les noms de ces arguments ainsi que le nom du projet (`groupId` et `artifactId`).*

2. **Attention**, cette classe `Main` sera utilisée dans les test unitaires JUnit servant à la validation de votre compilateur, ne la déplacez pas. Normalement, seule la méthode `compiler()` est à modifier au sein de cette classe.

ANTLR (`http://www.antlr.org`). Pour ne pas devoir écrire les analyseurs lexicaux et syntaxiques à la main, vous utiliserez ANTLR, un outil qui permet de générer un ensemble de classes Java effectuant ces analyses à partir d'une grammaire donnée. Le *template* Maven vous est fourni avec une grammaire incomplète et un exemple d'utilisation de celle-ci. Les premiers chapitres du livre de référence d'ANTLR [6] vous expliqueront comment encoder une grammaire et comment manipuler un arbre de syntaxe abstrait pour valider le code en entrée et générer le code en sortie. Nous vous conseillons également de lire la description du pattern 17 (*Symbol Table for Nested Scopes*) du livre de Terence Parr [5].

StringTemplate (`http://www.stringtemplate.org`). Enfin, StringTemplate est une bibliothèque Java utilisée pour générer du code à partir de *templates* (il s'agit du même mécanisme que celui utilisé en PHP, JSP, etc.). Celle-ci devrait vous faciliter la vie lors de la génération de code complexe.

3.8 Nous contacter

En ce qui concerne les questions, remarques et autres problèmes concernant le projet, nous vous demandons de vous conformer aux consignes suivantes :

- Vous ne comprenez pas un point de l'énoncé, vous avez un doute concernant un élément de la spécification, vous rencontrez un problème technique avec les outils de développement : postez un message sur le **forum** Webcampus adéquat, un assistant y répondra.
- Pour toute autre question, vous pouvez contacter par mail `james.ortizvega@unamur.be` ou au bureau 432.

Tous les mails devront comporter la mention "[S&S]" dans leur sujet afin d'être correctement filtrés. Veillez (comme il se devrait en toute circonstance) à être polis, respectueux, calmes et de bonne foi dans vos communications. En particulier, nous ne garantissons pas une réponse immédiate à vos messages. De plus, veillez à réfléchir par vous-même et à chercher la réponse à votre question dans les ressources à votre disposition (énoncés, documentation, livres [1, 3, 4, 5, 6], vos camarades...) avant de nous écrire.

Références

[1] S. Chacon and B. Straub. *Pro Git, second edition.* Apress, 2015. `https://git-scm.com/book/fr/v1`.

[2] M. Fowler. *Domain-specific languages.* Pearson Education, 2010.

[3] T. O'Brien, J. Casey, B. Fox, J. Van Zyl, J. Xu, T. Locher, D. Fabulich, E. Redmond, and B. Snyder. *Maven by Example.* Sonatype, 2015. `http://books.sonatype.com/mvnex-book/reference/public-book.html`.

[4] T. O'Brien, M. Moser, J. Casey, B. Fox, J. Van Zyl, E. Redmond, and L. Shatzer. *Maven : The Complete Reference.* Sonatype, 2015. `http://books.sonatype.com/mvnref-book/reference/public-book.html`.

[5] T. Parr. *Language implementation patterns : create your own domain-specific and general programming languages.* The Pragmatic Programmers, 2009.

[6] T. Parr. *The Definitive ANTLR 4 Reference.* The Pragmatic Programmers, 2013.