
04/01/2017

# Rapport : écriture d'un compilateur

*Compilateur Java*

Mathilde Prévost / Steve Négrine

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Table des matières

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Introduction

Ce projet a pour but de concevoir un compilateur simplifié, afin d'apprendre et assimiler les bases de la compilation.

Notre compilateur est développé en Java. Il transforme un code fourni (code source) dans un fichier texte en un pseudo code assembleur (code cible) qui est lui-même compilé par un assembleur / émulateur C permettant de gérer une pile de cases mémoire : Mini Stack Machine (MSM).

Notre compilateur suit les étapes suivantes : le code source fourni au format texte est analysé lexicalement par le Lexer qui est lui-même utilisé par le Parser, l'analyseur syntaxique. L'analyse de type est effectuée et les variables sont stockées dans la Table des Symboles. Un arbre syntaxique est alors généré et utilisé par le Generator, la phase de génération de code cible qui produit le code final souhaité : le code compilé. Ce code est ensuite fourni à la MSM et exécuté.

Nous avons travaillé étape par étape en développant chaque partie du compilateur : Lexer, Parser, Table des Symboles, Generator. Nous avons tout d'abord développé l'essentiel de ces différentes parties, afin d'avoir un compilateur fonctionnel. Nous avons ainsi pu tester chaque phase et nous nous sommes ainsi assurés de son bon fonctionnement. Nous l'avons par la suite, enrichie progressivement.

Nous avons donc inventé un langage de programmation très basique (inspiré du langage C et PHP) qui respecte une grammaire spécifique. Cette grammaire suit les contraintes fournies en cours. Celle-ci est décrite dans la partie suivante.

Nous allons dans ce rapport fournir une documentation du langage à compiler (code source) puis faire un bilan du compilateur et enfin conclure par un bilan personnel sur le déroulement de ce projet.

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Documentation du langage à compiler

Dans cette partie, nous allons fournir la grammaire utilisée par le compilateur et détailler la sémantique du langage de programmation à compiler.

### Grammaire

Voici les différents niveaux de notre grammaire :

- Niveau primaire : P <- ident | ident "(" E* ")" | cst_int | -P | (E)
- Niveau multiplicatif : M<- P (* M | / M | % M | ε)
- Niveau additif : A <- M (+ A | - A | ε)
- Niveau comparatif C <- A (==A | !=A | <A | <=A | >A | >=A | ε)
- Niveau expression E <- C
- Niveau fonction F <- Type IDENT "(" Args ")" I Création d'une fonction
- Niveau racine X <- I Noeud racine

- Niveau affectation Aff <- ident = E
- Niveau instruction I <- var ident : Type ; | Aff | if (E) I (else I | ε) | while (E) I | {I*} | for (Aff;E;Aff) | return E; | echo E;

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Sémantique du langage

Le programme doit obligatoirement commencer par une fonction main, qui est donc la fonction principale du programme. Toutes les fonctions doivent se terminer par un return sauf la fonction main. Nous nous sommes fortement inspirés du langage C pour écrire notre langage. On a un fonctionnement procédural avec une forte utilisation des fonctions.

Les espaces, tabulations et sauts de ligne n'ont pas d'impact sur le code. Ainsi une indentation est possible pour une meilleure lisibilité du code.

Tous les mots-clé (*var, int, if, else, while, for, return, echo*) sont insensibles à la casse.

### Les types

Notre compilateur ne supporte qu'un seul type, le type entier : int.

### Les variables

Toutes variables doit être déclarée au préalable avant d'être utilisée.

La syntaxe pour déclarer une variable *var* est la suivante :

```
var nom_var : type;
```

Considérons l'exemple suivant. Il déclare une variable i de type entier :

```
var i : int;
```

# Rapport : écriture d'un compilateur

*Compilateur Java*

## La boucle while

Le compilateur exécute l'instruction tant que l'expression de la boucle *while* est évaluée comme **TRUE**. La valeur de l'expression *E* est vérifiée à chaque début de boucle, et, si la valeur change durant l'exécution de l'instruction, l'exécution ne s'arrêtera qu'à la fin de l'itération.

Vous pouvez regrouper plusieurs instructions dans la même boucle *while* en les regroupant à l'intérieur d'accolades. S'il n'y a qu'une seule instruction dans la boucle, les accolades ne sont pas nécessaires.

La syntaxe des boucles *while* est la suivante :

```
while (E) {
    I*
}

Ou

while (E)
    I
```

Considérons l'exemple suivant. Il affiche les chiffres de 0 jusqu'à 10 :

```
var i : int;

while (i<=10) {
    echo i;
    i=i+1;
}
```

# Rapport : écriture d'un compilateur

*Compilateur Java*

## La boucle for

La première affectation *Aff1* est exécutée, quoi qu'il arrive au début de la boucle. Au début de chaque itération, l'expression *E* est évaluée. Si l'évaluation vaut **TRUE**, la boucle continue et les instructions sont exécutées. Si l'évaluation vaut **FALSE**, l'exécution de la boucle s'arrête. À la fin de chaque itération, l'affectation *Aff2* est exécutée.

Vous pouvez regrouper plusieurs instructions dans la même boucle *for* en les regroupant à l'intérieur d'accolades. S'il n'y a qu'une seule instruction dans la boucle, les accolades ne sont pas nécessaires.

La syntaxe des boucles *for* est la suivante :

```
for (Aff1; E; Aff2) {
    I*
}

Ou

for (Aff1; E; Aff2)
    I
```

Considérons l'exemple suivant. Il affiche les chiffres de 0 jusqu'à 10 :

```
var i : int;

for (i=0; i<=10; i=i+1) {
    echo i;
}
```

# Rapport : écriture d'un compilateur

*Compilateur Java*

## L'instruction if – else

L'expression *E* est convertie en sa valeur booléenne. Si la conversion vaut **TRUE**, les instructions dans le *if* sont exécutées. Si la conversion vaut **FALSE**, ce sont les instructions du *else* qui sont exécutées. Le *else* est facultatif.

Vous pouvez regrouper plusieurs instructions dans la même instruction *if else* en les regroupant à l'intérieur d'accolades. S'il n'y a qu'une seule instruction dans le *if*, les accolades ne sont pas nécessaires.

La syntaxe de l'instruction *if* est la suivante :

```
if (E) {
    I*
}

else {
    I*
}

Ou

if (E)
    I
else
    I
```

Considérons l'exemple suivant. Il affiche la valeur de a, si a est inférieur ou égal à b, et b sinon :

```
var a : int;

var b : int;

a = 5;

b = 7;

if (a<=b) {
    echo a;
}

else {

    echo b;

}
```

# Rapport : écriture d'un compilateur

*Compilateur Java*

## L'instruction echo

L'instruction *echo* permet d'afficher une variable (un int dans notre compilateur).

La syntaxe de l'instruction *echo* est la suivante :

```
echo x;
```

Considérons l'exemple suivant. Il affiche la variable a :

```
var a : int;

a = 5;

echo a;
```

## L'instruction return

L'instruction *return*, lorsqu'elle est appelée depuis une fonction, termine immédiatement la fonction, et retourne l'argument qui lui est passé.

La syntaxe de l'instruction *return* est la suivante :

```
return x;
```

Considérons l'exemple suivant. Il retourne 5, donc a (dans la fonction main) vaut 5 :

```
int main() {
   var a : int;
   a = nom_fonction();
}
int nom_fonction() {
   return 5;
}
```

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Bilan du compilateur

Le compilateur que nous avons développé propose les fonctions minimales souhaitées.
Nous allons présenter brièvement ce qui marche et ce qui ne marche pas.

### Ce qui marche

Notre compilateur gère les fonctions, les boucles *while* et *for*, ainsi que les conditions. Nous avons également implémenté les opérateurs de comparaisons et les opérateurs arithmétiques.

Nous avons ajouté une instruction pour afficher une variable (*echo*) qui est très pratique pour voir si notre programme fonctionne correctement (affichage du résultat et des calculs intermédiaires).

Pour nous aider à générer l'arbre syntaxique correctement, nous avons fait un affichage de celui-ci, dans la console.

### Ce qui ne marche pas

Le plus gros défaut de notre compilateur est le fait qu'il ne gère pas bien les priorités de calcul. C'est pour cela que nous utilisons des calculs simples dans nos exemples.

Les autres fonctionnalités de notre compilateur fonctionnent, d'après nos tests.

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Bilan personnel

Il est assez frustrant de ne pas pouvoir aller plus loin dans la conception d'un compilateur, par manque de temps. Nous aurions souhaité approfondir certains aspects, ajouter des fonctionnalités et améliorer sa performance.

Cependant, en aussi peu de temps, nous avons pu voir l'essentiel de la conception d'un compilateur. Certaines améliorations que nous aurions pu faire ne sont pas si complexes et le plus important est de comprendre le fonctionnement d'un compilateur ainsi que les difficultés dans sa réalisation.

En tant que développeurs et utilisateurs de différents langages de programmation, nous comprenons d'autant mieux l'importance d'un bon compilateur. On peut écrire un très mauvais code, le compilateur pourra corriger ses défauts. Nous avons pu constater, grâce à ce projet, que certaines bonnes pratiques de programmation sont très importantes et améliorent ainsi les performances d'exécution du programme.

On a pu constater qu'il est possible de rendre la vie du développeur plus simple en laissant les aspects difficultueux, proche de la machine, au compilateur et non au développeur. En plus de simplifier la vie du développeur, cela garantit une utilisation optimale de la machine. Par contre, ceci peut effectivement générer du code compilé très performant mais cela ajoute une complexité supérieure gérée au niveau du compilateur et augmente ainsi la durée de compilation.

## Les problèmes rencontrés

L'une des difficultés de l'exercice est de comprendre la signification exacte de chaque étape dans l'ensemble du compilateur, surtout en début de projet. Il ne fallait pas mélanger le rôle de chaque analyseur (lexical et syntaxique), bien comprendre le fonctionnement et l'utilité de la Table des Symboles. Vers la fin du projet, tout devient plus clair, les enchaînements des phases plus explicites. On a une vue de l'ensemble du compilateur bien définie avec une limitation des rôles plus précise. De plus, il était important de bien comprendre le fonctionnement de la pile pour écrire la phase de génération de code. Le rythme de deux heures de théorie puis deux heures de pratique est idéal pour la compréhension.

Nous avons passé beaucoup de temps sur la conception de la grammaire. Ce qui n'est pas étonnant puisque l'on a ajouté de nouvelles fonctionnalités au fur et à mesure. La compréhension du fonctionnement des variables a été l'un des aspects le plus délicat à appréhender. De même la notion de fonction n'a pas été des plus simples à mettre en place.

# Rapport : écriture d'un compilateur

*Compilateur Java*

## Ce que l'on aurait voulu faire de plus

Par manque de temps, nous n'avons pas pu implémenter tout ce que nous avions prévu en début de projet.
Nous voulions afficher l'arbre en json pour avoir un meilleur aperçu de la hiérarchie dans l'arbre, mais cela demandait trop de temps, donc nous avons mis ce développement de côté.
Nous avons laissé, dans le code, la possibilité d'utiliser le type String mais nous n'avons pas eu le temps de voir ce type en cours. Nous ne l'avons donc pas implémenté.

Nous aurions aimé pouvoir gérer les « a+=1 » à la place des « a = a + 1 ».

Pour finir, nous avons vu en cours comment gérer les tableaux de int, mais la partie génération de code nous a pris plus de temps que prévu, nous n'avons donc pas pu aller plus loin sur les types.

## Ce qui nous plaît / déplaît dans notre langage

Notre langage est élémentaire et facile d'utilisation. Nous avons essayé de rendre son utilisation la plus simple possible, en tenant compte de nos aptitudes à développer un compilateur.

Cependant, sur certains aspects, il est un peu trop simple. Par exemple, pour la déclaration et l'affectation, nous sommes obligés de faire deux lignes, et non une seule comme cela existe dans tous les langages.

Par ailleurs, nous sommes un peu frustré de ne pas gérer les priorité de calculs, nous sommes donc obligé de faire des calculs simples, en ne mélangeant pas les opérateurs arithmétiques.

## Conclusion

Nous sommes finalement assez satisfaits de notre compilateur. Ce projet nous a permis de comprendre parfaitement la chaîne de compilation d'un code source vers un code cible et l'importance d'avoir un bon compilateur tant pour la performance que pour la simplicité d'écriture d'un programme.