Rapport du Mini-Compilateur Java

Analyse Lexicale, Syntaxique et Sémantique

Abdelghani BOUKTIT

7 décembre 2025

Table des matières

Introduction

Ce projet consiste à créer un **mini-compilateur en Java** pour un petit langage simplifié inspiré du C. Le compilateur effectue :

- **Analyse lexicale (lexer)** : transformation du texte source en une suite de **tokens**.
- **Analyse syntaxique (parser)** : construction d'un **AST (Abstract Syntax Tree)** à partir des tokens.
- **Analyse sémantique** : vérification des déclarations de variables et des règles de cohérence.

L'objectif est de détecter les erreurs lexicales, syntaxiques et sémantiques avant toute exécution.

Structure du compilateur

Lexer (Analyse lexicale)

Le lexer transforme le code source en **tokens** (unités lexicales). Chaque token contient :

- **Type** : IDENTIFIANT, NOMBRE, AFFECTATION, PLUS, TANT_QUE, etc.
- **Lexème** : la valeur exacte du texte.
- **Position** : ligne et colonne dans le code source.

Exemple de tokens pour `x = 5 + 2;` :

```
IDENTIFIANT('x') à 1:1
AFFECTATION('=') à 1:3
NOMBRE('5') à 1:5
PLUS('+') à 1:7
NOMBRE('2') à 1:9
POINT_VIRGULE(';') à 1:10
FIN_DE_FICHIER('') à 1:11
```

Erreurs lexicales possibles :

- Caractères inconnus ($, @) → `Erreur lexicale : Caractère inconnu ...`
- Symbole `!` sans = → `Erreur lexicale : Symbole '!' inattendu sans '='.`

Parser (Analyse syntaxique)

Le parser transforme les tokens en **AST** (Arbre Syntaxique) représentant la structure du programme. Il vérifie la syntaxe du programme :

- Parenthèses et accolades équilibrées
- Points-virgules correctement placés
- Opérateurs avec opérandes valides

Exemple d'AST pour :

```
x = 5;
y = 10;
while (x < y) {
    x = x + 1;
}
```

```
BlockStatement
 AssignStatement(varName='x', expr=NumberExpr(5))
 AssignStatement(varName='y', expr=NumberExpr(10))
 WhileStatement
    condition: BinaryExpr(left=VarExpr('x'), operator=<, right=VarExpr('y'))
    body:
       BlockStatement
          AssignStatement(varName='x', expr=BinaryExpr(left=VarExpr('x'), operator=+,
```

Erreurs syntaxiques possibles :

- Expression incomplète (`x = 5 + ;`) → `Erreur de syntaxe : L'opérateur '+' n'a pas d'opérande à droite`
- Parenthèses ou accolades manquantes → `Erreur de syntaxe : Attendu PARENTH_FERM mais trouvé ...`
- Jeton inattendu dans une instruction → `Erreur de syntaxe : Jeton inattendu dans l'instruction ...`

SemanticAnalyzer (Analyse sémantique)

Le SemanticAnalyzer vérifie la cohérence des variables :

- Déclaration avant utilisation
- Pas de réutilisation incorrecte

Exemple correct :

```
x = 5;
y = x + 1;
```

- Pas d'erreur, x est déclarée avant utilisation.

Exemple incorrect :

```
y = x + 1;
```

- Erreur sémantique : Variable non définie : x

Cas test et résultats

Cas	**Code exemple**	**Résultat**
Valide simple	x = 5 ;	Compilation réussie
Valide avec while	while (x < 10) x = x + 1 ;	Compilation réussie
Erreur syntaxique	x = 5 + ;	Erreur de syntaxe : L'opérateur '+' n'a pas d'opérande à droite ...
Erreur lexicale	x = 5 $ 2 ;	Erreur lexicale : Caractère inconnu : ''...
Erreur sémantique	y = x + 1 ;	Erreur sémantique : Variable non définie : x

Parenthèse manquante	if (x < 5 x = x + 1 ;	Erreur de syntaxe : Attendu PARENTH_FERM mais trouvé ACCOLADE_OUVR ...

Principe de l'analyseur syntaxique

L'analyseur syntaxique lit les tokens générés par le lexer. Il reconstruit la structure hiérarchique du programme sous forme d'AST. Il applique des règles grammaticales du langage :

— Un `while` doit avoir une condition entre parenthèses et un bloc entre accolades
— Une assignation doit avoir un point-virgule à la fin
— Les expressions doivent être correctes (`x + y`, pas `x + ;`)

En cas d'erreur, le parser interrompt l'analyse et affiche un message précis avec ligne et colonne.

Conclusion

Ce mini-compilateur permet :

— De détecter les erreurs lexicales, syntaxiques et sémantiques.
— D'afficher des messages précis et en français, facilitant la correction.
— De représenter le programme sous forme d'AST, ce qui est utile pour un futur interpréteur ou générateur de code.