

Mini-rapport – Projet « Mini-compilateur C »

Auteur : khaldi abdeldjalil

Date : 8 décembre 2025

Dépôt GitHub : https://github.com/Abdeldjalil-David/TP_Compile-Project

1. Grammaire choisie

Sous-ensemble du langage C restreint aux constructions impératives classiques :

Program → TranslationUnit*
TranslationUnit → ExternalDecl
ExternalDecl → VarDecl | FunctionDecl | Statement
VarDecl → Type DeclaratorList ';'
DeclaratorList → Declarator (';' Declarator)*
Declarator → IDENT ('=' Expression)?
FunctionDecl → Type IDENT '(' ParamList? ')' CompoundStmt
ParamList → Param (';' Param)*
Param → Type IDENT

Statement → CompoundStmt | IfStmt | WhileStmt | ForStmt | ReturnStmt | ExprStmt | ';'
CompoundStmt → '{' Statement* '}'
IfStmt → 'if' '(' Expression ')' Statement ElsePart
ElsePart → 'else' Statement | ε
WhileStmt → 'while' '(' Expression ')' Statement
ForStmt → 'for' '(' ForInit? ';' ForCond? ';' ForPost? ')' Statement
ForInit → VarDecl | ExprStmt | ε
ForCond / ForPost → Expression | ε
ReturnStmt → 'return' Expression? ';'
ExprStmt → Expression? ';'

Expression (précédence croissante)

Assignment → LogicalOr (AssignOp Assignment)?
AssignOp → '=' | '+=' | '-=' | '*=' | '/='
LogicalOr → LogicalAnd ('||' LogicalAnd)*
LogicalAnd → Equality ('&&' Equality)*
Equality → Relational ('==' | '!=') Relational)*
Relational → Additive ('<' | '>' | '<=' | '>=') Additive)*
Additive → Multiplicative ('+' | '-') Multiplicative)*
Multiplicative → Unary ('*' | '/' | '%') Unary)*
Unary → '!' | '-' | '++' | '--') Unary | Postfix
Postfix → Primary ('++' | '--')*
Primary → IDENT | NUMBER | STRING | 'true' | 'false' | 'NULL' | '(' Expression ')'

//Types supportés : int, float, double, char, bool, void.

//Commentaires « // » et « /* ... */ » ignorés.

2. Analyseur lexical (Lexer.java)

- Parcours unique gauche-droite, sans backtrack.
- Reconnaissance par « maximal munch ».
- Erreurs : caractère inconnu, nombre mal formé, chaîne non fermée, & ou | isolés.
- Retour : List<Token> + liste d'erreurs lexicales.

3. Analyseur syntaxique (Parser.java)

- Descente récursive « Recursive Descent » : une méthode par non-terminal.
- Look-ahead limité à 3 tokens (peek) pour décider VarDecl vs FunctionDecl.
- Récursivité gauche éliminée, associativité opérateurs respectée.
- Récupération sur erreur : mode « panic » jusqu'au prochain « ; », « } » ou EOF ; on continue pour signaler le maximum d'erreurs en une passe.
- Pas de construction d'AST : seule validation syntaxique + rapport d'erreurs.

4. Structure du projet

```
mini-compilateur/  
├─ README.md  
├─ LICENCE  
└─ Main.java
```

5. quelque tests

a) Lexical

- on tape 3.14 → OK, on tape 1e10 → le lexeur râle (normal, pas encore géré).
- on met un « @ » dans le code → le lexeur crie « unknown char ».

b) Syntaxe

- un if/else bien écrit → passe.
- on oublie un « ; » après int x → message rouge « ';' expected ».
- un for(int i=0 i<10;i++) → même pas besoin de compiler, le parseur dit « t'as oublié un ; ».

c) Vrais petits programmes

- fact.c : fonction factorielle, ça compile sans souci.
- calc.c : mini-calculatrice en while/if, pareil, zéro erreur.

6. Compilation & exécution

```
$ javac src/Main.java
```

```
$ java -cp src Main
```

(coller le code source, terminer par une ligne contenant seulement « # »)

Sortie :

Parsing: no syntax errors.

ALL Correct 