

Mini-rapport – Projet « Mini-compilateur C »
Auteur : khaldi abdeldjalil
Date : 8 décembre 2025
Dépôt GitHub : https://github.com/Abdeldjalill-David/TP_Compile-Project

1. Grammaire choisie

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

Program	\rightarrow	TranslationUnit*
TranslationUnit	\rightarrow	ExternalDecl
ExternalDecl	\rightarrow	VarDecl FunctionDecl Statement
VarDecl	\rightarrow	Type DeclaratorList ;
DeclaratorList	\rightarrow	Declarator (, Declarator)*
Declarator	\rightarrow	IDENT (= Expression)?
FunctionDecl	\rightarrow	Type IDENT (ParamList?) CompoundStmt
ParamList	\rightarrow	Param (, Param)*
Param	\rightarrow	Type IDENT
Statement	\rightarrow	CompoundStmt IfStmt WhileStmt ForStmt ReturnStmt ExprStmt ;
CompoundStmt	\rightarrow	{ Statement* }
IfStmt	\rightarrow	'if' (Expression) Statement ElsePart
ElsePart	\rightarrow	'else' Statement ϵ
WhileStmt	\rightarrow	'while' (Expression) Statement
ForStmt	\rightarrow	'for' (ForInit? ; ForCond? ; ForPost?) Statement
ForInit	\rightarrow	VarDecl ExprStmt ϵ
ForCond / ForPost	\rightarrow	Expression ϵ
ReturnStmt	\rightarrow	'return' Expression? ;
ExprStmt	\rightarrow	Expression? ;

Expression (précédence croissante)

Assignment	$\rightarrow \text{LogicalOr} (\text{AssignOp Assignment})?$
AssignOp	$\rightarrow '=' '+=' '-=' '*=' '/='$
LogicalOr	$\rightarrow \text{LogicalAnd} (\text{' '} \text{ LogicalAnd })^*$
LogicalAnd	$\rightarrow \text{Equality} (\text{'&&'} \text{ Equality })^*$
Equality	$\rightarrow \text{Relational} ((\text{'=='} \text{'!='}) \text{ Relational })^*$
Relational	$\rightarrow \text{Additive} ((\text{'<' } \text{'>' } \text{'<=' } \text{'>=' }) \text{ Additive })^*$
Additive	$\rightarrow \text{Multiplicative} ((\text{'+'} \text{'-' }) \text{ Multiplicative })^*$
Multiplicative	$\rightarrow \text{Unary} ((\text{'*' } \text{'/' } \text{'%' }) \text{ Unary })^*$
Unary	$\rightarrow (\text{'!' } \text{'-' } \text{'++' } \text{'--' }) \text{ Unary } \text{ Postfix}$
Postfix	$\rightarrow \text{Primary} (\text{'++' } \text{'--' })^*$
Primary	$\rightarrow \text{IDENT} \text{NUMBER} \text{STRING} \text{'true'} \text{'false'} \text{'NULL'} (\text{ Expression })$

//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. quelques 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 