|  |
| --- |
| téléchargement (2).png |

République Algérienne Démocratique et Populaire Ministère de l’Enseignement Supérieur et de la Recherche **Université des Sciences et de la Technologie Houari Boumediene**

**Faculté d’Informatique**

**Département Informatique**

**Compiler project**

**Realisation of mini-compiler MinIng**

**Realized by:**

Khettaoui nahla

#### **1. Introduction to ANTLR**

**ANTLR** (ANother Tool for Language Recognition) is a powerful parser generator that is widely used to build languages, tools, and interpreters. It works by taking a formal grammar specification in the form of a .g4 file and automatically generates code for a lexer (tokenizer), parser, and sometimes a tree listener or visitor. It is particularly useful for building compilers and interpreters.

In this project, **ANTLR** was used to create the lexer, parser for the **MinING** language, which is the source language for our compiler.

#### **2. Steps to Use ANTLR in the Project**

To use **ANTLR** in a compiler project, we followed these steps:

##### **Step 1: Add ANTLR as a Dependency**

To use ANTLR in a Java project, the first step is to add it as a dependency in your build system. In our case, we are using **Maven** as the build tool. To include ANTLR in your pom.xml file, add the following dependency:

A screen shot of a computer program

AI-generated content may be incorrect.

**Step 2: Define the Grammar**

The core of the compiler's parsing process is the grammar that defines the structure of the **MinING** language. This grammar is written in ANTLR syntax and is stored in a .g4 file (e.g., MinING.g4) it contains the parser and the lexer. ANTLR is an **LL(\*)** parser, meaning it reads input left-to-right and uses arbitrary lookahead to resolve parsing decisions. This allows it to handle complex grammars and resolve ambiguities, unlike traditional parsers with fixed lookahead. It handles operator precedence by ordering grammar rules appropriately, ensuring higher-precedence operations are parsed first. ANTLR's powerful grammar design enables accurate parsing of complex languages without needing explicit precedence rules.

That’s why having this grammar won’t cause ambiguity or lack of precedence:

A computer screen with text

AI-generated content may be incorrect.

We defined all the nessecairy grammar/Tokens in the MinING.g4 file.

**Step 3: Generate Lexer and Parser**

Once the grammar is defined, ANTLR generates the lexer and parser using the following command: antlr4 MinING.g4

This will generate Java files (e.g., MinINGLexer.java, MinINGParser.java, MinINGBaseVisitor.java) that are used in Java code.

Those files are stored in the gen Folder

**Step 4: Write the Compiler Logic**

With the lexer and parser generated, we can now use them to process MinING source code. The lexer tokenizes the source code, while the parser processes the token stream according to the grammar rules.

**Step 5: Using the Visitor Pattern**

ANTLR provides an abstract base visitor class, MinINGBaseVisitor, which is used to define the actions to perform during the tree traversal. This pattern allows the separation of logic from the structure of the program, making the code more modular and maintainable.

**Grammar:**

**Lexical Elements**

Here, you should explain the tokens that the lexer will recognize. These include keywords, operators, data types, and symbols.

* **Keywords**: You define reserved words such as IF, ELSE, FOR, WRITE, READ, VAR\_GLOBAL, DECLARATION, INSTRUCTIONS, and CONST.
* **Operators**: The grammar supports logical operators (&&, ||, !), comparison operators (==, !=, <, >, <=, >=), and arithmetic operators (+, -, \*, /).
* **Symbols**: You define punctuation symbols such as parentheses, braces, semicolons, and commas, which are used to structure expressions and statements.
* **Data Types**: The language supports basic types: INTEGER, FLOAT, and CHAR.
* **Identifiers**: Variables are identified by the ID rule, which enforces that identifiers start with a capital letter followed by lowercase letters or digits.

**Syntax Rules**

Here, explain how the different components of the language are structured using syntax rules.

* **Program Structure (prog)**: The program begins with VAR\_GLOBAL, followed by DECLARATION, and concludes with the INSTRUCTIONS block. This defines the overall structure of a MinING program, where global declarations, local declarations, and instructions are enclosed within braces.
* **Declarations**:
  + The declaration rule allows declaring variables and constants. The TYPE keyword specifies the type of the variable, and the indexing rule allows for indexing (e.g., arrays) with optional initialization values.
  + Constants are defined with the CONST keyword, and you can initialize variables or constants with values.
* **Expressions**:
  + **Arithmetic Expressions (expr\_arith)**: Handles arithmetic operations such as addition, subtraction, multiplication, and division.
  + **Logical Expressions (expr\_logical)**: Defines logical expressions using operators like AND, OR, and NOT.
  + **Comparison Expressions (expr\_comparison)**: Allows comparisons using relational operators (<, >, ==, !=, etc.).
* **Control Structures**:
  + **Conditionals (condition)**: Defines the structure of IF and optional ELSE blocks. The condition in the parentheses is evaluated, and the appropriate block is executed based on the result.
  + **Loops (boucle)**: Defines FOR loops, which include an initialization expression, a condition for loop continuation, and an increment or step expression. This rule captures the basic structure of a FOR loop.
* **Instructions**:
  + The instruction rule captures various types of instructions such as assignments (affectation), conditions (condition), loops (boucle), input (entree), and output (sortie).
  + **Assignments (affectation)**: Assigns values to variables, which could be either arithmetic expressions or string literals.
* **Array Indexing (indexing)**: This rule allows variables to be indexed by expressions (e.g., x[5+2]), providing flexibility for handling arrays with dynamic indices.
* **Array Initialization (array\_init)**: Defines how arrays are initialized with expressions, which may be arithmetic expressions, ensuring flexibility in array definitions.

**Java Implementation:**

In this compiler, I extended the MinINGBaseVisitor class to implement various functionalities:

1. **Semantic Analysis**: I created a class, SemanticAnalysis, to handle semantic checks like type checking, variable declaration validation, and scope management. The visitor methods in this class check the validity of expressions and variable usages based on the rules defined in the language specification. Here's a summary of key methods:

* **visitDeclaration**: Handles variable and constant declarations, checks for type validity, redeclaration, and array initialization issues.
* **areTypesCompatible**: Compares declared types with the assigned value types.
* **visitArray\_init**: Handles array initialization.
* **visitInstruction**: Handles instructions.
* **visitSortie**: Checks output statements for valid expressions and declarations.
* **visitEntree**: Handles input statements, ensuring declared variables are used.
* **visitBoucle**: Validates loop counter variables, their types, and loop expressions.
* **visitExpr\_logical**: Ensures operands of logical expressions are of type BOOLEAN.
* **visitCondition**: Checks conditions in if statements, ensuring valid expressions and declared variables.
* **visitExpr\_comparison**: Validates operands of comparison expressions and returns BOOLEAN.
* **visitAffectation**: Ensures assigned variables are declared, not constants, and have compatible types

1. **Intermediate Code Generation**: The QuadrupletsGeneration class extends the visitor to generate intermediate code in quadruplet form. It traverses the parse tree, converts statements and expressions into quadruplets, and handles instructions like loops and conditionals.

**Methods**:

* **visitDeclaration**: Handles variable declarations, generates quadruplets for initialization (including string literals for arrays).
* **visitAffectation**: Manages assignments, including handling string literals and evaluating expressions.
* **Arithmetic Expressions**: Methods like visitMultDiv, visitAddSub, etc., generate quadruplets for arithmetic operations.
* **Logical Expressions**: Handles logical operations (AND, OR, NOT) by generating appropriate conditional jumps.
* **Control Structures**: For conditionals (visitCondition) and loops (visitBoucle), it generates quadruplets for condition checks and loop execution.
* **Input/Output**: visitSortie generates quadruplets for output operations and visitEntree handles input operations .

1. **Symbol Table Management**: A SymbolTable class is used to manage the declarations and types of variables. It ensures that variables are properly scoped and initialized, and it stores their corresponding attributes.
2. **Expression Type Checking:** In addition to general semantic analysis (such as symbol resolution), our compiler also ensures type correctness for expressions. To achieve this, we implemented an **ExpressionTypeChecker** class, which extends ANTLR's MinINGBaseVisitor. This class performs type checking for various expressions encountered in the parse tree.
3. **MinINGCompiler** class is our main class that processes MinING language code using ANTLR. It handles syntax and semantic analysis, followed by generating intermediate code in quadruplet form.

* **Error Handling:**
* **Syntactic Errors:**

Syntax errors are caught during parsing. The default error listeners are replaced with ConsoleErrorListener, which logs any syntax errors to the console. If there are syntax errors, the program halts further execution.

* **Semantic Errors:**

After parsing, the **SemanticAnalysis** visitor checks for semantic errors, such as undeclared variables or type mismatches. These errors are stored in an ErrorTable for better management. If errors are found, they are printed to the console.

* **Quadruplet Generation:**

If no errors are detected, the QuadrupletsGeneration visitor converts the parse tree into quadruplet intermediate code, which is then printed to the console.

**Tests and validation**

Here’s a test on the code

1. Declaration: we can declare in both scopes, we can initialise a string (table of chars) with a string, we can declare and initialise multiple vars at the time

A screen shot of a computer program

AI-generated content may be incorrect.

The scoped are stored correctly.

**Generated quads:** here’s the initialisation of the var/String (char tab)

A screenshot of a computer program

AI-generated content may be incorrect.

1. Instructions:A computer screen shot of a black screen with white text

   AI-generated content may be incorrect.

We can create nested loops/conditions, the semantic check is done (checking initialisations, types…etc)

the quads result:

A screenshot of a computer program

AI-generated content may be incorrect.

The comparison results are stored in a tmp variable that is either 0 or 1 and can be nested using logical operations