Project Report: Compiler for Java Subset

TY 71

- Rachit Nimje 12
- Rajnandini Dharashive 13
- Arya Rajvaidya 15
- Riddhi Shende 42

Guide - Prof. Sheetal Phatangare

Abstract—*This report presents the design and implementation of a basic compiler for a subset of the Java programming language. The compiler performs lexical analysis, syntax analysis, semantic analysis, and generates intermediate code (ICG). It parses a restricted set of Java constructs, including class definitions, main method declarations, variables, expressions, conditionals, and loops. The intermediate code generation phase produces a machine-independent representation of the program, which can be further optimized or translated into machine code. The paper discusses the methodology used to implement each phase of the compiler, the results obtained, and the potential for extending the compiler to handle more complex Java features.*

I. INTRODUCTION

Compilers are essential tools in the software development lifecycle, converting high-level source code into machine-readable instructions. This paper focuses on the creation of a basic compiler for a subset of the Java language, intended as an educational tool to demonstrate the core concepts of compiler construction. The compiler implemented in this project covers the following phases:

- **Lexical Analysis**: Tokenizing the input source code.
- **Syntax Analysis**: Generating a syntax tree based on the input tokens.
- **Semantic Analysis**: Creating a symbol table to track variable declarations and scopes.
- **Intermediate Code Generation**: Producing an intermediate, machine-independent code representation.

The subset of Java supported by this compiler includes class definitions, main method declarations, variable assignments, conditionals, and loops.
This report outlines the design and implementation of each phase of the compiler.

II. METHODOLOGY

2.1 Phase 1: Lexical Analysis

The first phase of the compiler involves **lexical analysis**, where the input Java subset program is scanned and divided into a sequence of tokens. Each token corresponds to a distinct language construct, such as keywords, identifiers, constants, and operators.

Example Tokens:

- T_CLASS: Represents the class keyword.
- T_ID: Represents identifiers like variable and class names.
- T_OParen: Represents an opening parenthesis (.
- T_CParen: Represents a closing parenthesis).
- T_NUM: Represents numerical constants.
- T_STRING: Represents string types.

In this phase, the lexical analyzer reads the input Java source code and outputs a stream of tokens that will be used by the syntax analyzer.

2.2 Phase 2: Syntax Analysis

The **syntax analysis** phase constructs a **syntax tree** based on the sequence of tokens provided by the lexical analyzer. It checks whether the sequence of tokens follows the grammar of the Java subset, ensuring that the program structure is valid.

The grammar used for this subset includes constructs such as class definitions, main method declarations, variable declarations, and conditional and iterative statements. The following grammar rules are used:

- class_def: modifier Class_head
- Class_head: T_CLASS T_ID T_OParen main_stmt T_CParen
- main_stmt: modifier modifier modifier T_MAIN '(' T_STRING '[' ']' T_ARGS ')' T_OParen stmts T_CParen
- modifier: T_PUBLIC | T_STATIC | T_VOID
- stmts: stmts stmt {$$=new_node("MStmts",$1,$2);} | stmt {$$=$1;}
- stmt: T_ID T_ASSG T_expr ';' | var_decl | cond_stmts | iter_stmts
- cond_stmts: T_IF '(' cond ')'
- iter_stmts: T_WHILE '(' cond ')' | T_FOR '(' var_decl cond ';' T_ID T_ASSG T_expr ')'
- T_expr: T_expr '+' T_expr | T_expr '-' T_expr | T_expr '*' T_expr | T_expr '/' T_expr | T_Const
- T_Const: T_NUM | T_ID
- cond: T_expr T_GEQ T_expr | T_expr T_LEQ T_expr | T_expr T_GE T_expr | T_expr T_LE T_expr | T_expr T_S_EQ T_expr
- var_decl: T_INT T_ID T_ASSG T_expr ';'

The syntax tree represents the hierarchical structure of the program. Each node in the tree corresponds to a construct in the grammar, such as a class definition, a statement, or an expression.

2.3 Phase 3: Semantic Analysis

In the **semantic analysis** phase, the compiler performs checks on the program's meaning. The semantic analysis ensures that the program is not only syntactically correct but also logically consistent.

One of the primary tasks during this phase is to build a **symbol table**, which is a data structure that keeps track of all the variables, functions, and their properties (such as types and scope) encountered during the parsing process.

For example:

- The symbol table will contain entries for each declared variable, with information like its type (e.g., int, string), whether it's in the correct scope, and whether it has been initialized.
- The semantic analyzer also checks for errors like undeclared variables, type mismatches in assignments and expressions, and function calls with incorrect parameters.

2.4 Phase 4: Intermediate Code Generation (ICG)

The **intermediate code generation** (ICG) phase is responsible for generating an intermediate representation of the program. This intermediate code is machine-independent and can later be translated into target machine code or bytecode.

The ICG is a sequence of simple operations, which can include assignments, expressions, and control flow operations. Each statement in the original program is translated into one or more intermediate code instructions.

Example Intermediate Code (ICG) for Subset Java:

```
e = 2
i = 0
t0 = i > e
t1 = not t0
if t1 goto L0
t1 = 12 * 4
t2 = t1 / e
t3 = t2 + 1
a = t3
d = 1
L0: a = 5
w = 0
t4 = w < e
t5 = w + 1
t6 = not t4
if t6 goto L2
L1: q = 1
goto L1
L2: t6 = a < 10
t7 = not t6
if t7 goto L4
```

```
L3: r = 1
goto L3
L4:
```

In this phase:

- **Expressions** such as 12 * 4 and t1 / e are broken down into simpler operations like multiplication, division, and addition.
- **Control flow** instructions like if, goto, and loop handling (for, while) are also generated.

This intermediate code serves as the bridge between the high-level program and the lower-level machine code.

Pseudocode for parser.y

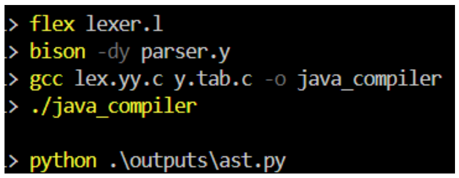
```
Main Program:
    Initialize symbol table
    Initialize files for:
        - Intermediate code output
        - Symbol table output
        - Syntax tree output

    Parse input file:
        For each statement:
            1. Build AST
            2. Update symbol table
            3. Generate intermediate code

    Print final symbol table
    Print AST structure
    Close all files
    Clean up memory
```

III. RESULTS AND DISCUSSION

The compiler has been tested on a set of Java subset programs, successfully tokenizing input source code, constructing syntax trees, checking for semantic errors, and generating intermediate code. Some of the results include:

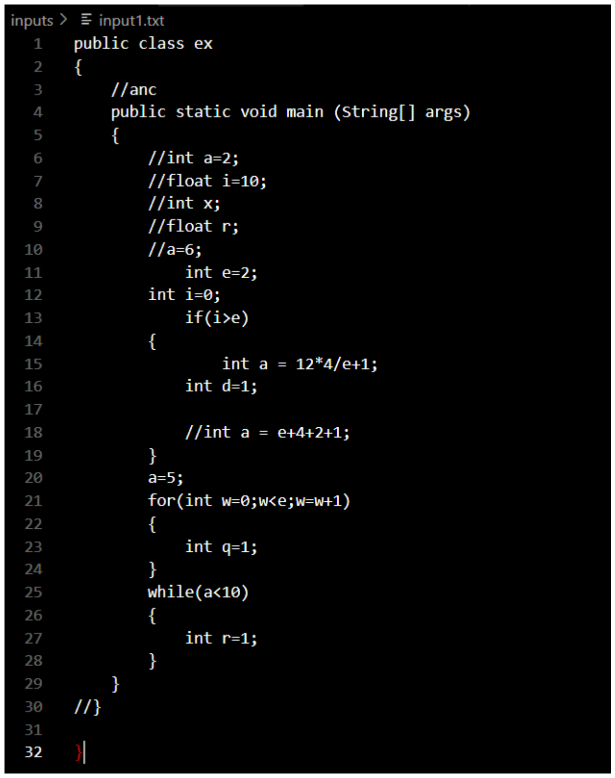
- **Tokenization**: The lexer correctly identifies keywords, identifiers, operators, and constants from the source code.
- **Syntax Tree**: The syntax analyzer generates valid syntax trees for all the supported Java constructs.
- **Semantic Analysis**: The semantic analyzer detects and reports errors such as undeclared variables and type mismatches.
- **Intermediate Code Generation**: The ICG phase produces correct intermediate instructions, which can be further processed for code optimization or machine code generation.

Fig.1. Execution steps



```
> flex lexer.l
> bison -dy parser.y
> gcc lex.yy.c y.tab.c -o java_compiler
> ./java_compiler

> python .\outputs\ast.py
```

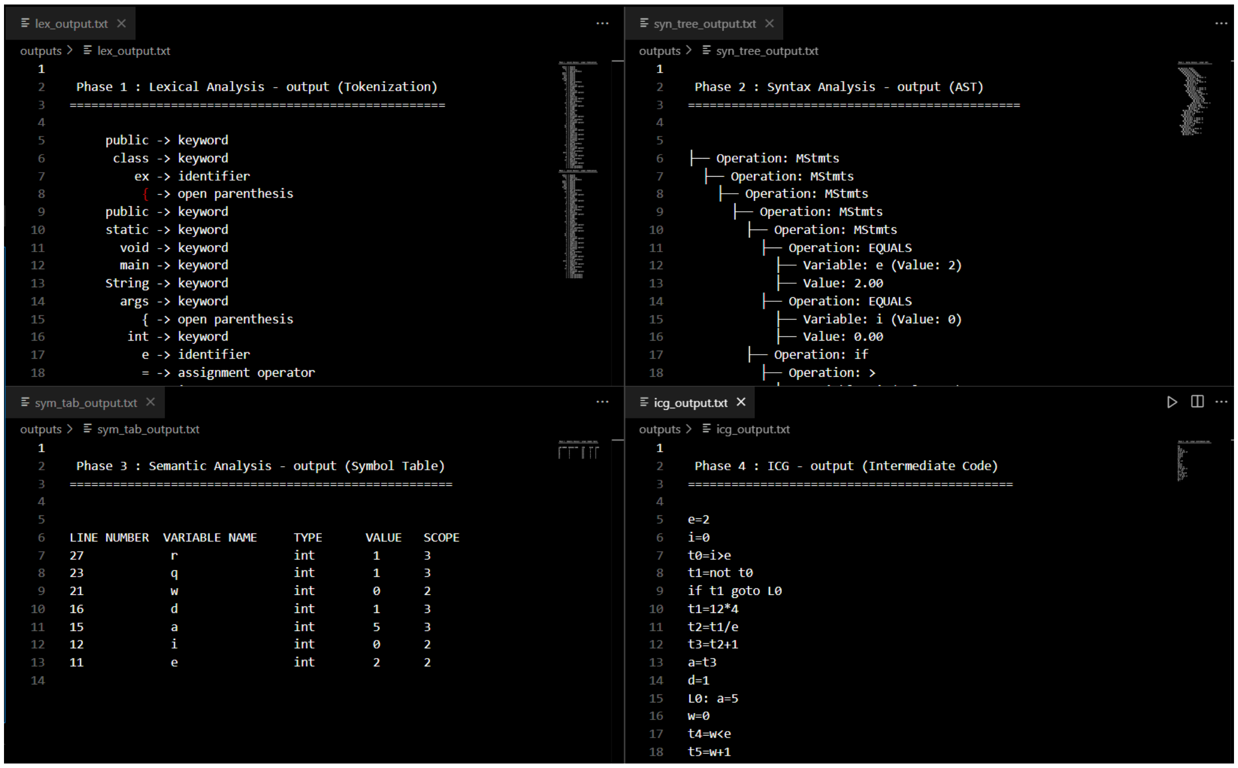
Fig.2. Input file



```
inputs > input1.txt
public class ex
{
    //anc
    public static void main (String[] args)
    {
        //int a=2;
        //float i=10;
        //int x;
        //float r;
        //a=6;
            int e=2;
        int i=0;
            if(i>e)
        {
                int a = 12*4/e+1;
            int d=1;

            //int a = e+4+2+1;
        }
        a=5;
        for(int w=0;w<e;w=w+1)
        {
            int q=1;
        }
        while(a<10)
        {
            int r=1;
        }
    }
//}

}
```

Output files

1. Lex_output
2. Syn_tree_output
3. Sym_tab_output
4. Icg_output
5. Ast_tree



```

 Phase 1 : Lexical Analysis - output (Tokenization)
=====================================================


     public -> keyword
      class -> keyword
         ex -> identifier
          ( -> open parenthesis
     public -> keyword
     static -> keyword
       void -> keyword
       main -> keyword
     String -> keyword
       args -> keyword
          { -> open parenthesis
        int -> keyword
          e -> identifier
          = -> assignment operator
```

```

 Phase 2 : Syntax Analysis - output (AST)
==============================================



├── Operation: MStmts
  ├── Operation: MStmts
    ├── Operation: MStmts
      ├── Operation: MStmts
        ├── Operation: MStmts
          ├── Operation: EQUALS
            ├── Variable: e (Value: 2)
            ├── Value: 2.00
          ├── Operation: EQUALS
            ├── Variable: i (Value: 0)
            ├── Value: 0.00
        ├── Operation: if
          ├── Operation: >
```

```

 Phase 3 : Semantic Analysis - output (Symbol Table)
=====================================================


LINE NUMBER  VARIABLE NAME      TYPE      VALUE    SCOPE
27                r             int       1        3
23                q             int       1        3
21                w             int       0        2
16                d             int       1        3
15                a             int       5        3
12                i             int       0        2
11                e             int       2        2

```

```

 Phase 4 : ICG - output (Intermediate Code)
=============================================


e=2
i=0
t0=i>e
t1=not t0
if t1 goto L0
t1=12*4
t2=t1/e
t3=t2+1
a=t3
d=1
L0: a=5
w=0
t4=w<e
t5=w+1
```

Fig.3. Output files

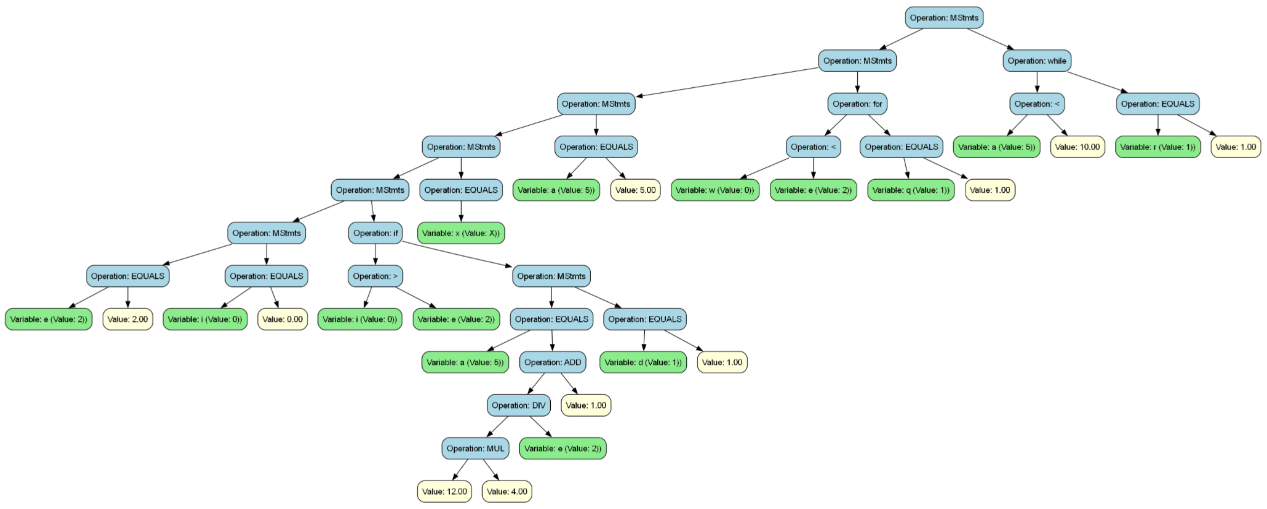


Fig.4. AST visualization

These results demonstrate the correctness and efficiency of each phase of the compiler. However, some limitations remain, such as the inability to handle advanced Java constructs like classes with methods, inheritance, or exceptions.

IV. FUTURE SCOPE

There are several opportunities for expanding and improving the current implementation:

- **Extended Syntax**: The current implementation supports only a limited subset of Java. Future work can extend the grammar to include more Java features like method declarations, object-oriented features (e.g., inheritance, polymorphism), and exception handling.
- **Error Handling**: The compiler could be enhanced to provide more detailed error messages during lexical, syntax, and semantic analysis phases, helping developers debug their code more easily.
- **Optimization**: The intermediate code generation phase can be improved by adding basic optimization techniques such as constant folding and propagation, loop unrolling, and dead code elimination.
- **Code Generation**: The final phase of the compiler, which converts intermediate code into machine code or bytecode, can be implemented to generate executable programs or Java bytecode (.class files).

V. CONCLUSION

In this project, we have presented a compiler for a subset of the Java programming language, implementing four key phases: lexical analysis, syntax analysis, semantic analysis, and intermediate code generation. The compiler successfully processes Java programs, identifies errors, and generates intermediate code, serving as a foundation for further improvements and extensions. This project provides a valuable educational tool for understanding the core principles of compiler construction.