

SEMANTIC ANALYSIS TOOL

A MINI PROJECT REPORT

Submitted by

PATURI SAI ROHIT [RA2011026010200]

KOUSHIKI RAY [RA2011026010199]

CATHY ANAND [RA2011026010216]

Under the guidance of

Dr. B.Pitchaimanickam

(Assistant Professor, Department of Computational
Intelligence)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

With specialization in AI and ML



SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

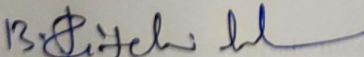


COLLEGE OF ENGINEERING &
TECHNOLOGY SRM INSTITUTE OF
SCIENCE & TECHNOLOGY S.R.M.
NAGAR, KATTANKULATHUR – 603

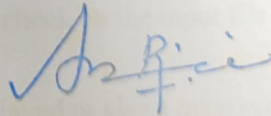
203

BONAFIDE CERTIFICATE

Certified that this project report “SEMANTIC ANALYSIS TOOL” is
the bonafide work of “PATURI SAI ROHIT [RA2011026010200],
KOUSHIKI RAY [RA2011026010199], CATHY ANNAD
[RA2011026010216]” of III Year/VI Sem B.tech(CSE) who carried out the mini project
work under my supervision for the course 18CSC304J- Compiler Design in SRM Institute of
Science and Technology during the academic year 2022-2023(Even sem).


SIGNATURE

Dr. B. Pitchaimanickam
Assistant Professor
Department of CINTEL


SIGNATURE

Dr. R. Annie Uthra
HEAD OF THE DEPARTMENT
Department of CINTEL

Abstract

The project consists of building a compiler for MiniJava, a subset of Java. MiniJava is designed so that its programs can be compiled by a full Java compiler like javac. Most things are well defined in the grammar or derived from the requirement that each MiniJava program is also a Java program.

- MiniJava is fully object-oriented, like Java
- MiniJava supports single inheritance but not interfaces.
- All MiniJava methods are “public” and all fields “protected”.
- In MiniJava, constructors and destructors are not defined

We will use the JTB tool to convert it into a grammar that produces class hierarchies. Then we will write one or more visitors who will take control over the MiniJava input file and will tell whether it is semantically correct, or will print an error message. The visitors built are subclasses of the visitors generated by JTB. In the end, we will have a Main class that runs the semantic analysis initiating the parser that was produced by JavaCC and executing the visitors. We will turn in the input grammar. The Main function parses and statically checks all the MiniJava files that are given as arguments.

Visitors

We use two visitors. The first one will create and initialize the symbol table using a bunch of java.util and user-defined data structures. This information will later on be

passed out to the next visitor, whose task is to type check on the input file. In case of a

flawless input, an appropriate message will be displayed or else a message describing

the kind and exact position, if possible, of the error.

• First Visitor

This visitor will implement the first pass on the input file, looking for errors.

To build a nice LookUp Table holding all information necessary for each class, an appropriate structure, ClassData, had to be defined, holding the meta data of class of

the source. So, we include Parent class name and a map with name-value pairs for each variable.

- **Second Visitor**

After constructing a Lookup Symbol Table, we need to traverse the syntax tree once

more and inspect errors. Any of the errors, will raise an exception. If no error is detected during the second traversal, the semantic correctness of the input program is unquestionable.

TABLE OF CONTENTS

S. No.	Title	Page. No.
1.	CHAPTER 1: Introduction	4
2.	CHAPTER 2: Requirements	5
3.	CHAPTER 3: Code 3.1 ClassData 3.2 FirstVisitor 3.3 Main 3.4 SecondVisitor 3.5 Error Handling 3.6 Triplet	6 - 17
4.	CHAPTER 4: Input and Output	18
5.	CHAPTER 5: Conclusion and Result	19
6.	CHAPTER 6: References	20

CHAPTER 1

INTRODUCTION

The purpose of this project is building a semantic analysis tool for a subset of Java, MiniJava, given the grammar it supports in JavaCC form and using JTB tool to convert it into class hierarchies. After that, a series of visitors will inspect the input file and report any possible semantic errors/mismatches.

Building a semantic analysis tool for MiniJava, given the grammar it supports in JavaCC form, using JTB to convert it into class hierarchies and a couple of visitors to detect possible semantic errors on the source file.

CHAPTER 2

REQUIREMENTS

For every MiniJava file, the program should store and print some useful data for every class such as the names and the offsets of every variable and method this class contains. For MiniJava we have only three types of variables (int, boolean and pointers). Ints are stored in 4 bytes, booleans in 1 byte and pointers in 8 bytes.

Directory tests contains all kinds of valid and invalid input files, taking into consideration each and every possible error type. Run bash script testing.sh under src directory to make sure all test cases are passed.

CHAPTER 3

CODE

- **Class Data**

ClassData.java

```
import javafx.util.Pair;
import java.util.ArrayList;
import java.util.Map;
import java.util.LinkedHashMap;

/* for each class, store some meta data */
public class ClassData{
    String parentName;
    Map <String, Pair<String, Integer>> vars;      // records of form: (variable_name,
    (type, offset))
    Map <String, Triplet<String, Integer, ArrayList<String>>> methods;  // records of
    form: (function_name, (return_type, offset, arglist))

    public ClassData(String parent){
        this.parentName = parent;
        this.vars = new LinkedHashMap<String, Pair<String, Integer>>();
        this.methods = new LinkedHashMap<String, Triplet<String, Integer,
        ArrayList<String>>>();
    }
}
```


- **First Visitor**

FirstVisitor.java

```
import java.util.Map;
import javafx.util.Pair;
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedHashMap;
import visitor.GJDepthFirst;
import syntaxtree.*;

public class FirstVisitor extends GJDepthFirst<String, ClassData>{

    /* use a map list storing (class_name, meta_data) pairs, also a list for each method
    holding its variables */
    public Map <String, ClassData> classes;
    private List <String> decls;
    private Integer row, nextVar, nextMethod;

    /* Constructor: initialize the map and row number */
    public FirstVisitor(){
        this.classes = new LinkedHashMap<String, ClassData>();
        this.row = new Integer(1);
        this.nextVar = new Integer(0);
        this.nextMethod = new Integer(0);
        this.decls = new ArrayList<String>();
    }

    /* Goal
    f0 -> MainClass()
    f1 -> ( TypeDeclaration() )*
```

```

f2 -> <EOF>
*/

public String visit(Goal node, ClassData data) throws RuntimeException{
    node.f0.accept(this, null);

    // visit all user-defined classes
    for(int i = 0; i < node.f1.size(); i++)
        node.f1.elementAt(i).accept(this, null);
    return null;
}

/* MainClass
class f1 -> Identifier(){
    public static void main(String[] f11 -> Identifier()){
        f14 -> ( VarDeclaration() )*
        f15 -> ( Statement() )*
    }
}
*/

public String visit(MainClass node, ClassData data) throws RuntimeException{
    /* get name of main method */
    String main = node.f1.accept(this, null);

    /* create a ClassData object and pass it down to the variables and methods
    declaration sections */
    ClassData cd = new ClassData(null);

    /* initialize offsets */
    this.nextVar = this.nextMethod = 0;

    /* pass ClassData to each variable of the variables section */
    for(int i = 0; i < node.f14.size(); i++)

```

```

        node.f14.elementAt(i).accept(this, cd);

    /* pass ClassData to each statement of the statements section */
    for(int i = 0; i < node.f15.size(); i++)
        node.f15.elementAt(i).accept(this, cd);

    this.classes.put(main, cd);
    return null;
}

/* TypeDeclaration
f0 -> ClassDeclaration() | ClassExtendsDeclaration() */
public String visit(TypeDeclaration node, ClassData data) throws RuntimeException{

    /* initialize offsets for each new class*/
    this.nextVar = this.nextMethod = 0;
    node.f0.accept(this, null);
    return null;
}

/* ClassDeclaration
class f1 -> Identifier(){
    f3 -> ( VarDeclaration() )*
    f4 -> ( MethodDeclaration() )*
}
*/

```

- **Main**

Main.java

```
import syntaxtree.*;
import java.util.Map;
import java.util.ArrayList;
import javafx.util.Pair;
import java.io.*;

class Main {
    public static void main (String [] args) throws RuntimeException{
        boolean displayOffsets = false;
        FileInputStream fis = null;
        ArrayList<String> paths = new ArrayList<String>();

        for(String arg: args){
            if(arg.equals("--offsets"))
                displayOffsets = true;
            else
                paths.add(arg);
        }

        /* for each file path given from the cmd */
        for(String path : paths){

            /* try and: open, parse and visit the syntax tree of the program */
            try{

                fis = new FileInputStream(path);
                MiniJavaParser parser = new MiniJavaParser(fis);
```

```

        System.out.println("\033[1m" + path +
"\033[0m\nParsing:\u001B[32m\033[1m Successful \u001B[0m");

        /* first, traverse the tree once, to create the lookup symbol table and report
minor errors*/

        FirstVisitor v0 = new FirstVisitor();
        Goal root = parser.Goal();
        root.accept(v0, null);

        System.out.println("First Pass Semantic Check:
\u001B[32m\033[1mSuccessful \u001B[0m");

        /* then traverse it one more time, to type check */
        SecondVisitor v1 = new SecondVisitor(v0.classes);
        root.accept(v1);

        System.out.println("Second Pass Semantic Check:
\u001B[32m\033[1mSuccessful \u001B[0m");

        System.out.println("Semantic Check: \u001B[32m\033[1m Successful
\u001B[0m");

        if(displayOffsets){
            System.out.println("Offsets\n -----");

            /* For each class */
            for (Map.Entry<String, ClassData> entry : v0.classes.entrySet()) {
                String name = entry.getKey();
                System.out.println("Class: " + name);

                /* For each variable of the class, print offset */
                System.out.println("\n\tFields\n\t ---- \n");

```

```

                                for(Map.Entry<String, Pair<String, Integer>> var :
entry.getValue().vars.entrySet())
                                System.out.println("\t\t" + name + "." + var.getKey() + ": " +
var.getValue().getValue());

                                /* For each pointer to a member method, print offset */
                                System.out.println("\n\tMethods\n\t----- \n");
                                for(Map.Entry<String, Triplet<String, Integer, ArrayList<String>>> func
: entry.getValue().methods.entrySet())
                                System.out.println("\t\t" + name + "." + func.getKey() + ": " +
func.getValue().getSecond());
                                }
                                }
                                }
                                /* handle exceptions */
                                catch(SemError e){
                                System.out.println(e.getMessage());
                                }
                                catch(ParseException ex){
                                System.out.println(ex.getMessage());
                                }
                                catch(FileNotFoundException ex){
                                System.err.println(ex.getMessage());
                                }

                                /* clean things up */
                                finally{
                                try{
                                if(fis != null) fis.close();
                                }
                                catch(IOException ex){

```

```
        System.err.println(ex.getMessage());
    }
}

}

if(!displayOffsets)
    System.out.println("Main: To view field and method offsets for each class rerun
with --offsets");
}
}
```

- **Second Visitor**

SecondVisitor.java

```
import java.util.Map;
import javafx.util.Pair;
import java.util.ArrayList;

import visitor.GJNoArguDepthFirst;
import syntaxtree.*;

public class SecondVisitor extends GJNoArguDepthFirst<String> {
    private Map<String, ClassData> table;
    private ArrayList<Pair<String, String>> scope;
    private ArrayList<String> args;
    private int index;
    private Integer row;
    private String className;

    /* add all inherited fields in scope of a derived class */
    private void addInScope(){
        Map<String, Pair<String, Integer>> map = this.table.get(this.className).vars;
        for(String key: map.keySet())
            this.scope.add(new Pair<String, String>(key, map.get(key).getKey()));
    }

    /* check whether a data type is a primitive or a user-defined class */
    private boolean isPrimitive(String type){
        return type.equals("integer") || type.equals("array") || type.equals("boolean");
    }

    /* recursively inspect whether a class is equivalent or sub-type of another */
```



```

private boolean isSubType(String childType, String parentType){
    String subType = childType;
    ClassData cd = this.table.get(subType);

    /* if it is a primitive type it should be identical */
    if(isPrimitive(childType))
        return childType.equals(parentType);

    /* else if it is user-defined it might can also be a sub-type */
    do {
        if(subType.equals(parentType))
            return true;
        subType = cd.parentName;
        cd = this.table.get(subType);
    } while (cd != null);
    return false;
}

/* return the last declared type for a given variable name */
private String lastDeclarationOf(ArrayList<Pair<String, String>> list, String
varName){
    for(int i = list.size()-1; i >= 0; i--){
        if(list.get(i).getKey().equals(varName))
            return list.get(i).getValue();
    }
    return null;
}

/* constructor */
public SecondVisitor(Map<String, ClassData> table) {
    this.table = table;
}

```

```

        this.row = new Integer(1);
        this.scope = new ArrayList<Pair<String, String>>();
        this.args = null;
    }

```

```

/* Goal
f0 -> MainClass()
f1 -> ( TypeDeclaration() ) *
f2 -> <EOF>
Visit all methods of the input file
*/

```

- **Error Handling**

SemError.java

```

public class SemError extends RuntimeException{
    private static final long serialVersionUID = 1L;

    public SemError(String message, int row) {
        super("\u001B[31m\u033[1mSem Error: " + message + " [ line:" + row + "
        ]\u001B[0m");
    }
}

```

- **Triplet**

Triplet.java

```
public class Triplet<T, U, V> {  
  
    private final T first;  
    private final U second;  
    private final V third;  
  
    public Triplet(T first, U second, V third) {  
        this.first = first;  
        this.second = second;  
        this.third = third;  
    }  
  
    public T getFirst() { return first; }  
    public U getSecond() { return second; }  
    public V getThird() { return third; }  
}
```

CHAPTER 4

INPUT

```
class A{
    int i;
    boolean flag;
    int j;
    public int foo() {}
    public boolean fa() {}
}
```

OUTPUT

```
A.i : 0
A.flag : 4
A.j : 5
A.foo : 0
A.fa: 8
B.type : 9
B.k : 17
B.bla : 16
```

CHAPTER 5

CONCLUSION AND RESULT

Our project titled “Semantic Analysis Tool” is successfully implemented and the desired outputs and results are thus obtained on the used test cases. The program performed semantic analysis on all files given as arguments and returned the correct expected outputs.

CHAPTER 6

REFERENCES

- [1] Reinhard Wilhelm; Helmut Seidl; Sebastian Hack (13 May 2013). Compiler Design: Syntactic and Semantic Analysis. Springer Science & Business Media. ISBN 978-3-642-17540-4.
- [2] Grune, D., Bal, H.E., Jacobs, C.J.H., Langendoen, K.G.: Modern Compiler Design. John Wiley & Sons (2000)
- [3] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. Addison-Wesley (2007)