Language and Traslator - Parser Report

Francesca Daniele
Carmelo Gugliotta
Marco Leto

A.Y. 2023/24

Our Semantic Analysis implementation respect and satisfy all the "project statement" requirments. Our approch to verify the semantic of the code consist in two steps of analysis on the AST that was been produced from the Parser.

1 SymbolTable

First of all we implemented the class `SymbolTable.java`, this data structur has the task of store all the information that need to be consulted during the different steps. Each SymbolTable have the following structure:

```
protected SymbolTable previousTable; //link to previous table
protected Map<String, SymbolTableEntry> entries;
```

Where the SymbolTableEntry is an abstract class that permise us to implement different Entry for each case. Particularly we have:

- `SymbolTableType`, used for the component where is needed to store just the type
- `SymbolTableProcedureType` used to memorize in the SymbolTable the procedure declaration. Each entry is characterized by:

  ```
  private final Map<String, SymbolTableEntry> parameters;
  private final SymbolTableType return_type;
  ```

  `parameters` will contains all the parameters of the procedure. As Map we us used the `LinkedHashMap` that permise use to store the order of the parameter for the future check.
- Considering that, we could have different procedures with the same name but different configuration, we implement the class `SymbolTableProceduresEntry` that has an unique parameter, `LinkedList<SymbolTableProcedureType>`, and store all the procedures with the same name.
- `SemanticStructType` used to store the Struct instantiation, and have the following parameters:

  ```
  private SymbolTable fields;
  private Type type;
  ```

- `SemanticArrayType` used to store the array instantiation, and store size and type of that:

  ```
  private int size;
  private Type type;
  ```

When we refer to `Type` we refer to the type we implemented in the Parser, that permise to give to the different project parts a global sense, making them communicate with each other.

2 Visitor Design Pattern

For the realization of the SemanticAnalysis we have used the Visitor Design Pattern, and we implemented 3 different Interface:

- `Visitable`: that defines an accept method that takes the visitor as an argument.Each visitable element class will implement this interface

```
public interface Visitable {
    void accept(Visitor visitor, SymbolTable symbolTable,
                SymbolTable structTable) throws SemanticException;
    Type accept(VisitorType visitor, SymbolTable symbolTable,
                SymbolTable structTable) throws SemanticException;
}
```

- `Visitor`: that must declare a method for each type of element in the data structure that we want to visit.
- `VisitorType`: that must declare a method for each type of element in the data structure that we want to get the Type

The class that implements `Visitor` are `SymbolTableUpdater` e `SemanticAnalysisVisitor` while `TypeCeckingVisitor` implements `VisitorType`.

3 First Step Check

The class `SemanticAnalysis.java` inizialize the two structures used to the analisys:

- `globalTable`
- `structTable` used to store just the structs, where each entry is an instantiation of `SemanticStructType`

and implement two important `public` method . The first one is `intilizeSymbolTable` that iterate on the AST calling on each node the `.visit` method, declared (as previously seen) in the `Visitable` class.

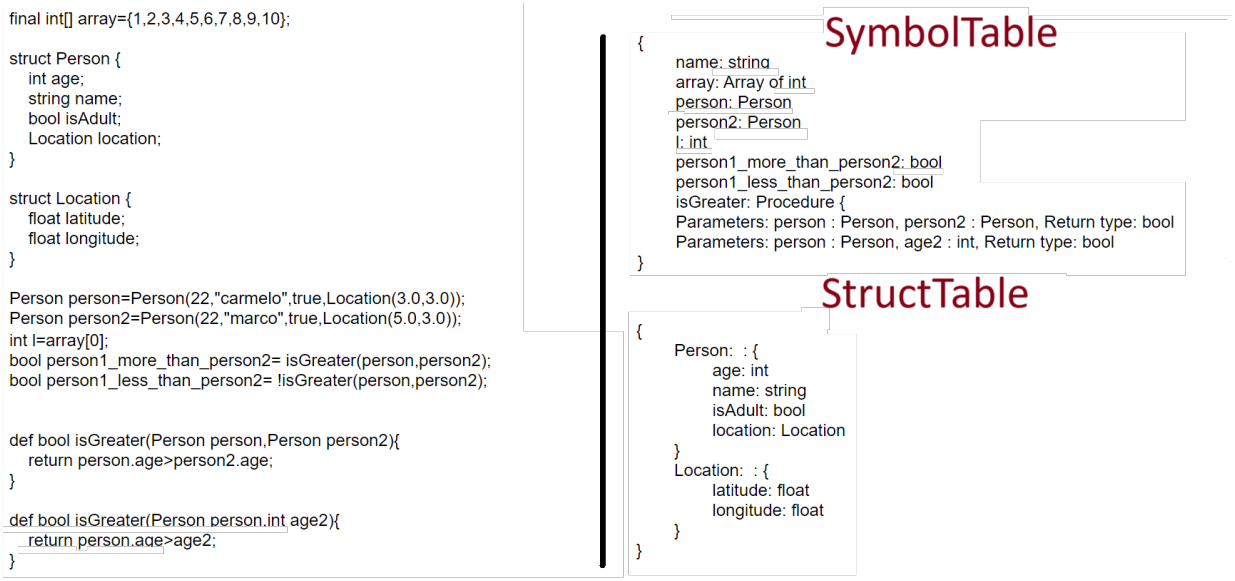
```
 public void intilizeSymbolTable(Program program) throws SemanticException {
     addBasicProcedure();
     Iterator<ASTNode> it= program.iterator(); //Si itera sull'AST creato dal
Parser
     while(it.hasNext()){
         ASTNode next=it.next();
         if(debugSemanticAnalysis){
             LOGGER.info("Adding to the symbol Table :\n\t\t "+next);
         }
         next.accept(symbolTableUpdater,globalTable,structTable);
     }
     if(debugSemanticAnalysis) {
         LOGGER.info("Symbol Table and Struct Table after the initialization");
         LOGGER.info("Symbol Table: \n" + globalTable);
         LOGGER.info("Struct Table: \n" + structTable);
     }

 }
```

In the first pass on the tree we just:

- Add constants to the symbol table beeing sure that there are not others with the same name
- Add stucts to the struct table checking that there are not others struct with the same name
- Add global variables to the symbol table checking that there are not others global variables with the same name
- Add the procedures statement adding to a list all the methods with the same name but different parameters

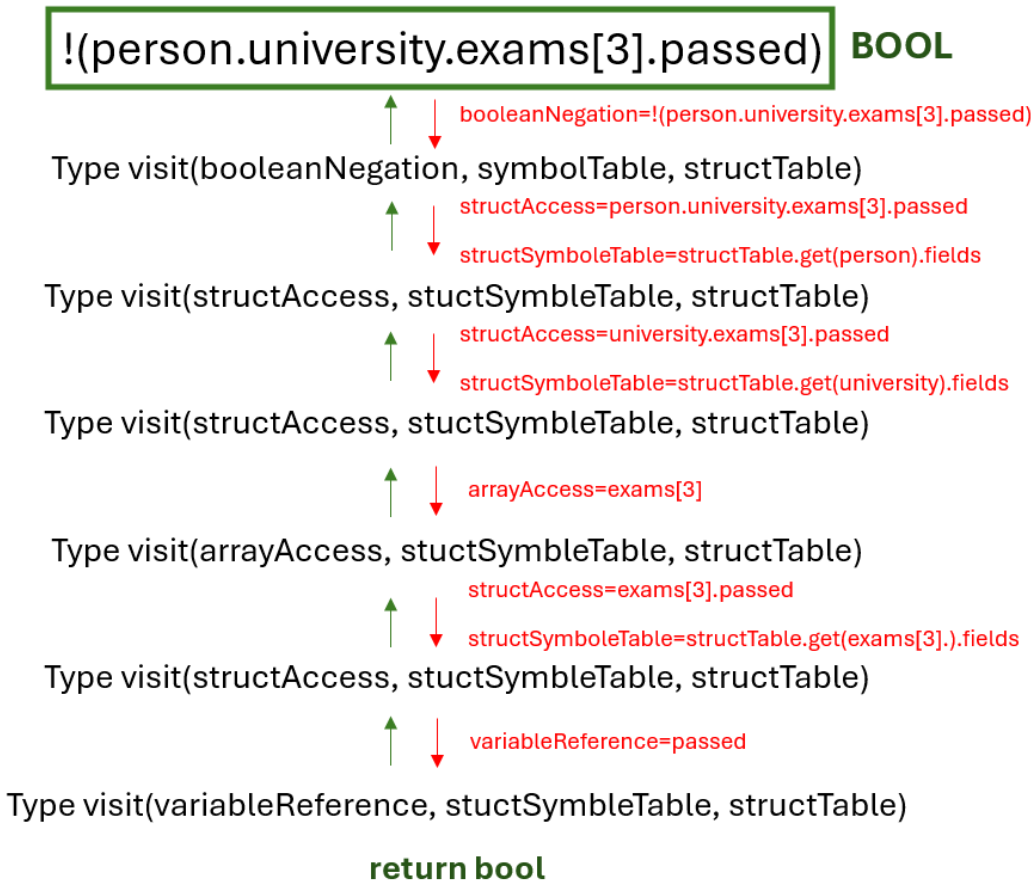
So, supposing to pass an example code, finished this part we have the following result:



4 Second Step Check

The second public method is `performSemanticAnalysis` that have the same structur of `intilizeSymbolTable` but used the methods implemented in the `TypeCeckingVisitor` that returs a `Type`. This becouse in this phase we have to check that all what we have already added to the tables don't present a Type Errors.
For this reason the main role of this step is to check that the value assigned to the variable (that are `ExpressionStatement`) match the type founded in the declaration. The logic that permise us to do this is recursive and work thanks to the definition of all the `visit` method defined in the `VisitorType`. In fact, except for `Value` and `VariableReferences` visit methods, all the other contains a call to an another `visit` method.

Supposing that we have to found the type of the following expression:



5 Test Class

At least we create the class `TestSemantic.class` that gives the possibility to see all the possible errors that Semantic is able to detect, using the following code sample:

```
public void TestStructErrors() throws ParserException, IOException {
        String input=". . .";
        String expected_error="Type Error: ...";
        Reader r=new StringReader(input);
        SemanticAnalysis s = new SemanticAnalysis(r, false, false, false);
        try{
            s.performSemanticAnalysis();
            fail("Should Fail");

        }catch (Exception e) {
            if(!e.getMessage().equals(expected_error)){
                fail("Error message is not correct "+ e.getMessage()+"\nExpected:
   "+expected_error);
            }
        }
    }
```