

DV1655/6 - ASSIGNMENT 2

SEMANTIC ANALYSIS

Suejb Memeti

Blekinge Institute of Technology

January 31, 2022

1 Introduction

In this assignment you are going to :

1. Construct the symbol table (described in Section 2), and
2. Perform the semantic analysis (type checking) (described in Section 3)

Accordingly, this assignment is split into two parts, (1) Constructing the Symbol Table and (2) Semantic Analysis.

The goal of the first part is to traverse the AST to construct the symbol table, which is a data structure that contains information related to identifiers, such as name, type, and scope.

The goal of the second part of this assignment is to traverse the AST and use the information stored in the symbol table to perform the semantic analysis.

The examination of this project is done through demonstrations during the lab sessions. It is expected that you submit the source code in Canvas. The source code should be compressed in a zip/tar file.

The (soft) deadline for demonstrating this assignment is the last lab session for assignment 2, i.e., February 24, 2022.

The solution should be implemented using C or C++, and it should be compiled and executed correctly on a Unix-based operating system.

1.1 Laboratory groups

You are encouraged to work in groups of two students. Groups larger than two is not accepted. Groups of one student are accepted, but not encouraged. The project is designed for groups of two students.

Discussion and help between laboratory groups are encouraged. However, you should be careful to not cross the border to plagiarism, see section 1.3 below.

1.2 Lecture support

In Canvas (the course management page), you should be able to find the Semantic Analysis lecture, which is related to this assignment. In that lecture we introduce the general concepts and theory for semantic analysis. We briefly describe the tree traversal algorithms and their role with respect to the construction of the symbol table and semantic analysis. We also describe the role of the symbol table and semantic analysis. Furthermore, we provide practical examples of how the symbol table is created and how it can be used for semantic analysis.

1.3 Plagiarism

All work that is not your own should be properly referenced. If not, it will be considered as cheating and reported as such to the university disciplinary board.

2 Part 1: Constructing the symbol table

2.1 Problem Description

The goal of this part of the assignment is to traverse the AST to construct the symbol table (ST). The ST is a data structure for storing information about the identifiers, such as the type and scope. Note that, in MiniJava, we have three types of identifiers: *class identifiers*, *method identifiers*, and *variable identifiers*. Each of those types represent the scope of such identifiers.

It is common for an identifier to have different type or value on a different scope in the program. So, the implementation of the symbol table needs to keep track of the current scope when analyzing our program. It needs to provide functionality to enter and exit scopes, add new identifiers to the symbol table, as well as functionality to lookup a given identifier.

Every time we start a new traversal of the AST, we need to reset the ST, hence we need to provide such functionality. For more implementation details on how we can construct a symbol table I suggest listening to the Semantic analysis video lecture.

2.1.1 Tasks to Complete

You are supposed to do the following:

- Write a tree traversal algorithm, which will visit all the nodes of the AST. Note that we already have a tree traversal algorithm implemented for the print-tree function, which basically visits each node of the tree and prints their name and type.
- Design a data-structure for the symbol table, which is able to store all the necessary information for all of identifiers.
- Perform a single left-to-right tree traversal and populate the symbol table.
- For debugging and demonstration purposes, write a method that prints the symbol table (the name and the type for each record in the symbol table).

2.1.2 Recommended approach

The recommended approach for constructing the symbol table is as follows:

- For each type of the identifiers (e.g., classes, methods, and variables) use a different type of records in the symbol table. A record represents an identifier in the symbol table. I therefore suggest a hierarchy of record types.
- The symbol table should be able to deal with the scopes of the identifiers. For instance, the variables defined inside a class could be accessed from anywhere in the class, whereas identifiers defined inside a method could only be accessed from that method. Note that in MiniJava, we can only define variables at the beginning of a method, hence we do not consider the cases where we define new variables inside the body of the *if-else* and *while* statements (in such cases we would need a nested level of scopes).
- Note that the symbol table can be constructed using a single left-to-right traversal of the AST.
- Note that the construction of the ST is concerned only with the declarations of identifiers (including variables, methods, and classes) in our program. The way those identifiers will be used is considered in the semantic analysis phase.
- Write a method for printing the symbol table and use it to verify the correctness of your solution. An example of the symbol table for one of the programs in the set of valid Java classes is provided in slide 14 of the Semantic Analysis lecture.

3 Part 2: Semantic analysis

3.1 Problem Description

The semantic analysis verifies the semantic correctness of the program. The main task of the semantic analysis include type checking, checking if identifiers are declared before they are used, checking for duplicate identifiers, and others.

The semantic analysis uses the information in the symbol table to perform the above tasks. It is expected for the semantic analysis to be implemented as a separate phase of the compiler, which means that you need to perform another traversal of the AST.

While the MiniJava grammar allows inheritance and polymorphism, in order to keep the assignment manageable within the given time frame the semantic analysis disregards inheritance and polymorphism. This means that the "TreeVisitor.java" test file in the set of valid test programs is expected to produce some semantic errors.

3.1.1 Tasks to Complete

You are supposed to do the following:

- Write another tree traversal function for performing the semantic analysis.
- Note that you may need to perform multiple tree traversals for different aspects of the semantic analysis. Also, note that the semantic analysis requires evaluation of types of sub-trees in a post-order fashion.
- Verify that all identifiers are declared and that there are no duplicate identifiers.
- Perform type checking for expressions, statements, methods, method calls, array accesses, and so on.

3.1.2 Recommended approach

For this assignment, I suggest to start with the functionality that checks for undeclared identifiers, as this is the simplest case of semantic analysis. For example, start checking if a variable of a method is declared inside the same class. Then you may extend the functionality to check for function calls outside the current class, and so on.

Then, you may start performing the type checking analysis. You may start with the simplest cases and then move to the more complex parts. For instance,

- Expressions: the type of all terminals inside an expression should be the same. For example, in ***a+b***; both *a* and *b* should be integers. ***10 + false*** should report a semantic error.
- Statements: check that the left-hand-side and right-hand-side of assignment statements are of the same type; check that the condition inside *if-statements* and *while-statements* is *Boolean* type; check that the type of the expression inside the print-statement is of type integer;. For example, in ***a=b***; type of *a* and *b* should be the same.
- Method declaration: check that the return type of the method is in accordance with the declared return type. In ***int a(int c)... return b***; the type of return expression *b* should be the same as the type of the method declaration (*int*).
- Method calls: verify that the number of parameters and the type of parameters inside a method call matches the number and type of parameters of the method declaration. For example, in ***a(x)***; the type of *x* and *c* should be the same as in their definition (see previous item).
- Array access: check that the expression inside the ***int []*** is of type integer; verify that the left hand side of the expression that has the *.length* member is an array of integer. For example, in ***b[a]***; ***b.length***; the type of *a* must be integer; the type of *b* must be *int[]*.
- and more ...

4 Testing

A set of valid test programs is provided by the Cambridge website. Use this set as a starting point. Extend this set of classes with your own test cases. A list of things you can use to test your compiler with is provided below:

4.1 Valid tests

- What happens if we return a class type?
- What happens if we use a method call inside the condition expression of an *if-statement*?
- What happens if we return an array of integers?
- What happens if variable foo is in both class A and B?
- ...

4.2 Invalid tests

- What if we use a method call as a statement?
- What if we have a void method?
- Method/Variable/Class duplication?
- Method/Variable/Class undefined?
- Undefined types?
- *.length* member on integers or boolean types?
- *boolean a; int b; b = b*a; if(a && b)*
- *int a[]; boolean b; a[b] ?*
- ...

5 Examination

During the demonstration:

- Your compiler should be easily compiled using the Makefile.
- You should be able to explain briefly how you have constructed the symbol table and how specific type checking steps work. For example, "Show me the code related to type checking of return statements?". Show me the code that does ...
- Have a set of valid test classes and invalid test classes ready. During the demonstration, you should use those test classes to demonstrate that the compiler is able to show semantic errors for the invalid examples list in Section 4. **Failing to demonstrate particular functionality will be considered as not-implemented.**
- You should print the symbol table and explain the information and how they relate to the input source code.