CS4201

Practical 1 - Semantic Analysis

190004947

October 22, 2021

University of
St Andrews

Word Count: 2,641

Contents

1 Introduction **1**

2 Using ANTLR4 to Generate Files **1**

3 Producing Parse Trees **1**

4 My Symbol Table Implementation **2**

5 Building the Symbol Table **3**

6 Semantic Analysis Phase **4**

7 Conclusion and Build Recipe **6**

1 Introduction

My implementation of a semantic analyser runs in two phases. Once ANTLR has produced a parse tree for the MiniJava program that is being type-checked, the generated visitor class is used to walk through the tree building up a symbol table of classes, methods, and variables using the generated listener class with my own modifications. The symbol table stores the types returned my methods and the types of each variable to be looked up in the second phase of the semantic analyser.

In phase two of the analysis, the same walker is used with a different listener class to check that the statements and expressions in the MiniJava program follow the type rules of the language. The type rules are implemented in my listener class when statement tokens are encountered in the parse tree. It will check that the code's semantics are in order by pattern matching with the grammar and looking up identifiers in the symbol table to see if the actual types match the expected types.

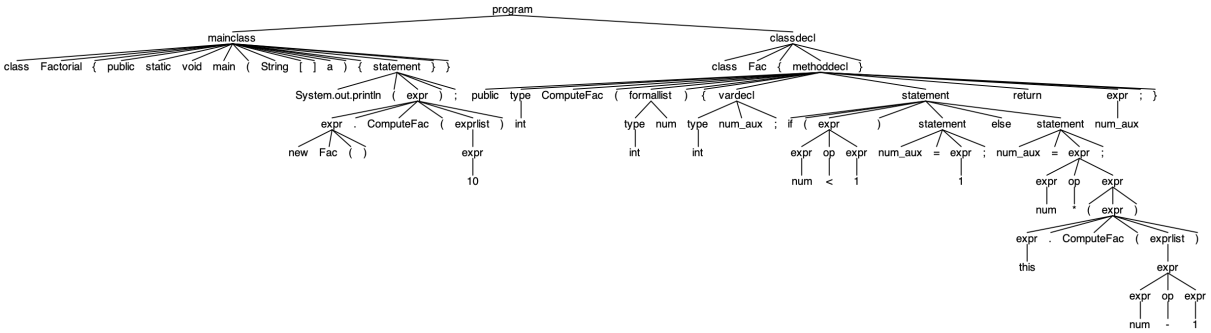
I would like to note that my final implementation is not functional and does not correctly run the semantic analysis on MiniJava programs. Instead, it will print out the correctly built symbol table. In light of this, in this report I will outline what the implementation is supposed to achieve and perhaps why my program does not achieve this.

2 Using ANTLR4 to Generate Files

To start my program, I created a file called MiniJavaGrammar with the .g4 extension and copied and pasted the grammar in from the file on StudRes. After this, I opened a terminal session and entered the root directory of the project to run 'antlr4 -visitor MiniJavaGrammar.g4'. This command invokes ANTLR4 to generate class files that will be used to produce a parse tree of a program from the grammar, walk through the parse tree, and listen to events in the walker. The '-visitor' tag tells ANTLR4 to build the walker class.

3 Producing Parse Trees

Once these files had been generated we can build and view parse trees from input MiniJava programs stored with the .txt. extension by running 'grun antlr.MiniJavaGrammar program tests/Test.txt -gui &' in the terminal. Using the Factorial program from StudRes as an example, the parse tree produced looks like this:



4 My Symbol Table Implementation

A core part of a semantic analyser is the symbol table which is a data collection storing the relevant details of every declared identifier. My implementation of the symbol table consists of four Java classes under a package called symboltable. The four classes I made are Program.java, Class.java, Method.java, and Variable.java.

The Program class has two attributes: a list of Class objects and a list of Method objects. The constructor here simply instantiates these two lists. Initially there was just a list of Class objects since in MiniJava you cannot have methods outside of a class, however, to aid in scope checking I later added the second list. Additionally, the Program class has two methods, one to add to the Class list and one to add to the Method list.

My Class class has a String attribute to store the ID of the class which will be used later for look up in the symbol table. As well as the ID, Class also stores a list of Variable objects and a list of Method objects. This list of Variable objects will keep a record of the variables declared at the beginning of the class outside of any methods, this will be crucial to access separately from method variables when scope checking. Again, we have methods for each List attribute which are used to add elements. In this class, and in the Method and Variable classes, I have overridden the toString() method in order to return the value of the id attribute.

Next is the Method class. It has five attributes including a String to store the id, a list of Variable objects that are the parameters of the method being declared, another list of Variable objects to store the local variables from the scope inside the method, a String to store the result type of the method which will be used in type checking, and lastly I added a Class type attribute that stores the parent class of the method. This last one is optional but I included it to allow easy access to the parent scope when running semantic analysis inside a method. The constructor in this class instantiates the local variables list and sets all other attributes from the parameters. These parameters are passed in by the first pass of the visitor which will be discussed in the next section.

Lastly, the Variable class will store a String for id like the others along with the type of the variable in a String for simplicity. On top of this I added a parent class attribute like in the

Method class and also a parent method attribute. The parent method attribute is set to null when instantiated and will stay this way if the Variable object is storing a class-wide variable that is outside a method, otherwise it will store the Method object of its' parent.

I decided to store my symbol table in this way as it was the simplest structure to visualise when accessing scopes. The hierarchy of Program-Class-Method-Variable is highly intuitive and the way each are stored as a member in the parent object's attributes makes it easy to search for specific member objects. A faster but slightly more involved implementation could've used a hash map to access the symbol table objects using their id as a key. This also would have allowed for some nice tricks when evaluating scope.

5 Building the Symbol Table

Once the classes in the symboltable package had been implemented, I can use those objects to build up a representation of the symbol table. To do this I downloaded the MiniJavaMain program from StudRes will will run the generated files to take the input file in, turn it into an input stream that ANTLR can recognise, automatically create a lexer for the program, use the lexer to tokenise the MiniJava example, build a parse tree using these tokens alongside the grammar, build a walker to traverse the parse tree, create an instance of the MiniJavaListener that I modified, and then walk the tree with the listener waiting for events.

In the MiniJavaListner class I added a global instance of Program that will essentially store the symbol tree using the member hierarchy of the classes outlined the the previous section. This variable is instantiated when the walker starts walking the parse tree and the listener executes the enterProgram method. The rest of my implementation in this file exists entirely within the enterDeclaration methods, this is because this pass is just to fill up the symbol table.

The simplest of the listener functions is enterMainclass and enterClassdecl because they simply instantiate a new Class object, set the ID to either "main" or the declared id, and adds the object to the Class list of the global Program variable.

When the parse tree walker encounters a method declaration it will have to visit a number of nodes to collect data for the Method object constructor before creating the Method object and adding it to it's parent Class object's Method list. This data includes an id, a list of Variable objects which are parameters, the result type, and the parent Class object that will be used in scope checking. The function checks if the method declaration has any parameters by visiting the node that it would expect parameters and if it finds a LPAREM token then there are none and otherwise it will find a FORMALLIST token. It will then call getChildCount on this token which will tell the program how many parameters the method has. With this information it will iterate over the parameters storing them in a list to be included in the constructor for the Method object. Once this is done, the result type is easily found by visiting the node described by the grammar pattern that stores the result type. The same is then done to find the ID of the parent class, after which an iterator will search through the global Program variable for

that class id and store the Class object with that id in the parentClass attribute of the method. When all this data has been collected and the Method object has been declared, it is added to the it's parent Class object in the Program variable.

The last method I implemented in the MiniJavaListener class is the enterVardecl method. Here it is very easy to collect the id and type for the Variable object constructor. After the Variable object has been created, the method must find where to insert it into the symbol table. There are two potential cases, either the variable is a class-wide variable or it is a local variable to some method. This can be checked with an if statement that uses the declaration's depth in the parse tree to work out which case we are in. In both cases, the listener will then search through the Program object to see if the variable has already been declared, in which case an error will be thrown saying that a variable with that id in the same scope has been declared. If it is a new variable declaration, the Variable object is added to either the Class's list of variables or the Method's list of variables depending on which case we ended up in from the earlier check.

Once the walker has completed a full pass of the parse tree, the listener will have entered all relevant data into the symbol table ready for the semantic analyser.

6 Semantic Analysis Phase

The next part of the program is the semantic analysis phase. This section of my implementation does not work and so I will outline what I have done and how it should work.

For this section I build out another listener class and instantiated it in the MiniJavaMain file before calling walker.walk() on it to run through the parse tree again, this time using the second listener. This listener, called MyListener, starts by declaring the global Program object (that will be collected from the first listener using a getter method) along with currClass and currMethod variables. These will be used to store the current scope. Initially both set to null, when entering a class, the currClass variable will be updated to store the Class object that the walker is currently inside and similarly for the currMethod variable. To check if identifiers called are in scope we first see whether or not we are inside a method and then we search through the relevant list of Method or Variable objects to find if it is within scope. This is achieved using a helper method I wrote called inScope().

The type-checking should then occur within the enterStatement method in the listener since that is where the MiniJava programs will be accessing expression tokens which include identifiers which must be matched to typing rules. The first thing this method will do is check which branch of the grammar we are entering by using the context parameter 'ctx' to access the children tokens. By checking for the existence of unique tokens we can determine which branch we are in. Since my implementation didn't work, this pattern matching switch is present with comments to show which branch we would be entering but the body of the if statement is empty.

The way I intended on type-checking the MiniJava programs is by using the resolveType function

I wrote. I actually wrote two versions of this function, the first of which is commented out and the other is left uncommented in the listener. The first version is how the type checking should be run and the second version was my attempt at skipping the type-checking of expressions and returning expected types in order to run type-checking on the statement tokens. For this reason, I will be describing the logic begind my original resolveType function.

The function takes in one parameter of type ParseTree called expression which stores the call to an expression token along with all of it's children. Just like in the enterStatement method, we open with pattern matching to find out which branch of the grammar we are entering. This is achieved by using the getChildCount function to see how many tokens are in the expression and then checking for unique tokens to identify the exact branch. Once the branch has been determined we know what kind of expression we are evaluating the types of, and (apart from the expr op expr production rule) we also know what types we are expecting each expression token to resolve to. For the expr op expr production, we can determine the expected types from the operation token. For example, if the operation is '' then we would expect both operands and the result type to be boolean while if the operation was '¡' we would expect the operands to be ints. When the function is trying to resolve the type of an ID token, it will call my getType helper method that will search through the symbol table and return the type associated with that identifier, be it a method result type or a static variable type.

The resolveType function is recursive since expressions (often) contain expressions. That is, the production contains the expr non-literal on both the left and right side of the rule. The base cases of the recursion is when the expression token only has one child; a terminal node in the parse tree. When this is true, we know that we can derive the expression type without any further calls to resolveType. To check if the type is an int I used a try-catch block that will attempt to use the Integer class' parseInt method on the string and store the expression type as an int if the method call is successful. If not, no error will be produced but the function will move onto the next base case which simply checks if the token value is "true" or "false" in which case we can conclusively say the expression type is boolean. The next base case finds the "this" token which tells the type-checker that we are now searching for a variable outside the method but inside the class. Lastly, we have an else block that means we have found an ID token, in which case the getType method is called and the type associated with that ID in the symbol table will be returned. The last type that we havent yet accounted for is the int array type. The reason for this is that our base cases are for when the expression token only has one child and an int array call will have four children as determined by the grammar branch "expr LSQUARE expr RSQUARE". To determine if an expression is of type int array, first we will need to match this grammar pattern and then we will need to type check the recursive expression calls by asserting that the first expr token has been declared with type int array and the second expr token resolves to an int.

When a grammar pattern has been found, the proceeding expression calls will be evaluated and compared to the expected types. In the case that an expression resolves to a type different to the expected type, an error should be printed telling the user what the issue was and on which line it occurred. After the error has been printed out, the expected type should be returned so that the semantic analyser can continue to check the rest of the program.

7 Conclusion and Build Recipe

When running my implementation of the type checker, many type issues were printed to the console when none should have been. I think that this was either caused by my resolveType function having incorrectly structured recursive calls or an issue in the way that the tree nodes are checked since accessing nodes using getChild and getParent can get very tricky.

Although my type checker didn't run correctly, I have made my best attempt in this report to explain my reasoning behind implementation decisions I made and the functional intentions behind these decisions. I hope that I have demonstrated a thorough understanding of the problem and how a potential solution would work.

In order to build my submission, set your classpath to the antlr-4.7.2-complete.jar file required to invoke ANTLR4 methods and run 'javac minijava/*.java' in the src directory. Then, to run the resultant program use the command 'java minijava.MiniJavaMain "tests/testfile.txt"' wehre MiniJavaMain is the entry point of the program and the second argument is a path to the MiniJava program to be parsed.