CS4201 Practical 1

190018035

October 12, 2022

Contents

1 Introduction

For this practical, I was asked to implement the process of type checking MiniJava, a subset of the Java programming language. This involved building a symbol table, checking that each record in the program is in the correct scope and finally type checking each statement and expression within the program. I was able to successfully create a scope checker and type checker by making use of multiple passes of the abstract syntax tree to create a valid symbol table. This scope checker and type checker passes 33 of my own tests along with the MiniJava tests provided in the tests/ folder.

2 Design & Implementation

Type checking MiniJava has a process of the two phases: building the symbol table and conducting the type checking. My implementation has a total of four passes, with the first three passes ensuring an accurate symbol table that checks thoroughly for scoping issues and the last pass purely for type checking each of the statements and expressions in the MiniJava program. In this section, I will detail what each of these passes do along with major design decisions I took whilst implementing them.

2.1 Creating the Data Structures

Before building the symbol table, I had to decide how I wanted to represent each of the records in the symbol table. Since every record has to have an identifier and a type, I made a Record class that would hold that information. This class would serve as a superclass for all other types of records that a symbol table can hold. The following are types of records that extends from this base record:

2.1.1 VarRecords

Variable records are simple records that only require the Record superclass's methods and global variables. This type of record is the smallest type of record in a program, holding only a identifier and an type.

2.1.2 MethodRecords

In a method's scope usually resides either a list of parameters and a list of variables local to that method. Method records in turn store this information in the form of a hashmap, with a hashmap for its parameters and a hashmap for its variables, where each parameter and local variable are represented as VarRecords. Hashmaps were used to store this information due to its efficient lookup capabilities. The one drawback with hashmaps, however, is that retrieving records from a hashmap are not in order. This becomes challenging when considering the fact that the arguments in method calls have to be in the same order as the formal paramters of the method. Therefore, I also made use of a ArrayList to keep track of the types of the parameters so as to preserve its order.

2.1.3 ClassRecords

In a class's scope are usually a list of global variables and a list of methods. Similar to the MethodRecord, the ClassRecord stores a hashmap of MethodRecords and a hashmap of global variables represented as VarRecords. Since it extends the Record class, its type is always given as "class".

2.1.4 ProgramRecords and the SymbolTable

The ProgramRecord acts as the root of the tree of data structures presented above. This root holds a hashmap of ClassRecords. While this class could be abstracted away in the SymbolTable class, where the SymbolTable holds a hashmap of classes, I decided to create this kind of record for readability and modularity. The SymbolTable class in turn holds a single ProgramRecord responsible for adding all ClassRecords in the program.

2.2 Pass 1: Building the Symbol Table

My first pass, implemented in the file SymbolTableBuilder.java, builds the basic structure of the symbol table. It only focuses on populating the symbol table based on variable, method, class and program declarations. In order to keep track of the current scope of the program, I made use of two global variables: a ClassRecord 'currClass' and a MethodRecord 'currMethod'. With these two global variables, I could keep track of which scope the program is currently in, with classes as the parent scopes and ProgramRecord keeping track of each class. Each declaration ensures that the current method and classes are updated if they need be. If a scoping error is found, an error message is printed to the console and the program is terminated. The result of this pass is a basic symbol table with all records accounted for.

There were a few considerations I took during this pass of type-checking. One consideration was the aspect of variable shadowing, where a variable in the inner scope has the same name as the variable in the outer scope. In my implementation, shadowing is permitted. That is, if a variable is declared as a global variable and redeclared within a method of the same class, that variable is bounded to the method and takes precedence over the global variable. Another consideration I had was for inheritance. In order to verify the inheritances in the next pass, I needed to keep track of whether each class declared extends another class, which was recorded with the global variable "parentClassId" within the ClassRecord structure.

2.3 Pass 2: Checking for Inheritance

The second pass, implemented in the file InheritanceListener.java, checks whether each of the classes populated in the symbol table from the previous pass has a parent by checking their "parentClassId" initialized in the first pass. If a class has a parent, every method and global variable of the superclass is added to the child class. Why check for inheritance in a separate pass instead of combining it with the first pass? Because class declarations in the first pass can be out of order, which means that if a superclass declaration only appears after the child class declaration, the program will throw an error as it doesn't recognize the superclass. The result of this pass is a symbol table that accounts for connections between classes, including added methods and global variables for child classes.

One major consideration I had in this pass was the possibility of cyclic inheritance. This meant that if class A extends B and class B extends A, the program should throw an error. Similarly, longer cycles of inheritance - such as A extends B, B extends C, C extends A - should also be identified and dealt with. To ensure there were no cycles of inheritance, I kept track of the chains of parents for each class until a parentClassId was null. I stored this sequence in an ArrayList, and before I checked the next superclass of a class, I checked that this ArrayList did not already contain the next superclass. If it did, the program printed an error and terminated.

Another consideration I had was variable shadowing in terms of inheritance. In my implementation, I allowed for child classes to have global variables with the same name as global variables in their superclass. The benefit of this implementation is that is is consistent with my design choice for shadowing in the first pass. The drawback, however, is that the child class will not be able to access the global variable of the superclass as the grammar doesn't allow for the "this" operator to be used on variables. One workaround for this would be for the superclass to have methods that manipulate its own global variable, which the child class can use if it needs access to the superclass global variable.

Lastly, I had to consider the case of overriding methods from superclasses. My implementation supports overriding, where if a call already has a method of the same name as its superclass, it does not add the superclass method to its map of records. This follows the same design principles as variable shadowing, taking the child class as the inner scope and the superclass as the outer scope.

2.4 Pass 3: Scope Checking Statements and Expressions

The third pass, implemented in the file ScopeCheckingStatements.java, evaluates statements and expressions to check whether they are bounded to the correct scope. Similar to the first pass, this pass has global variables "currClass" and "currMethod" to keep track of the current scope. In this pass, these variables are initialized using the symbol table from the previous passes. At the end of

this pass, all scoping issues should have been accounted for and a symbol table is printed to the console.

The purpose of this pass is to mainly check for two cases: 1) Unknown variables and 2) Inheritance checks. In each of the statements and expressions, variables that do not appear in the symbol table cause the program to throw an error and terminate. This is because the symbol table produced from the previous passes have account for all records in the program, included variables and methods that are inherited from other classes. Since no higher scope exists, these unknown variables are certain to not belong in the program.

The other case this pass has to check for is inheritance. Since all child classes now contain new methods and global variables inherited from their respective superclass in the previous pass, this pass can now start to check scopes for all the global variables and methods for a given class. For example, this pass can now check whether global variables exist within formal parameters.

2.5 Pass 4: Type Checking Statements and Expressions

The fourth and final pass, implemented in the file TypeChecker.java, evaluates statements and expressions to see if the types of each are consistent within the program. By this pass, the symbol table should be fully built, ensuring that every record that appears in the program is in the correct scope and belongs in the program. Therefore, there is no scope checking in this pass.

To implement the type checker, I made use of a stack that would push and pop Type objects (more on that later). The reasoning behind the stack can be boiled down to a simple case: if we have the expression "1 + 2", a stack would add both literals onto the stack. When the parser identifies the operator "+", the stack can pop off the two most recent items, where it can then be checked whether both literals are integers. Once verified, the stack can then push on another int, since the result of the operation should be an integer. In a more complicated case (e.g 1 + 2 + 3 + ...), the stack would perform well as it always holds the types of the last few elements of the expression before evaluation.

Another aspect to consider for type checking is the order in which Type elements should be pushed on to the stack. In a simple case such as 1 + 2, the order in which the parser enters the expression doesn't really matter as it can easily pick up the literals in the expression. However, consider the case of a method call with a potentially large list of parameters. In this case, it would be easier for the stack to start from right to left, evaluating the last few parameters till the start of the method call as it is certain that there is an end to the expression. Hence, instead of entering statements and expression as done in previous passes, I decided to make use of exiting statements in order to get the stack to evaluate expressions from the right to the left.

Unlike the previous few passes, where errors would result in the termination of the program, I had to ensure that the program continued even if an error occurred during a type check of an statement or expression. To do this, I made sure that even if there was a mismatch between types in an expression, the program still added the expected resulting type to the stack in order to continue along the pass without any cascading errors.

The last consideration I had for implementing this pass was method call expressions. These kind of expression were tricky as they demanded additional checks apart from type checking. This is best

seen in the case of parameter size and order. In order to type check a method call, I have to first make sure that number of arguments matches the number of formal parameters a method has. I also have to check that the arguments appear in the correct order in order to type check between the formal parameters and the arguments. To do this, we must know what is the actual method being called from the object rather than just evaluate its types. Therefore, I decided to make a Type object which would simply hold the type of a record as a String in all cases except ClassRecords. In the case of ClassRecords, both the record type and the actual ClassRecord itself is stored. By doing this, when an object calls a method, we can simply lookup the ClassRecord from the Type and find the method that corresponds to the method call.

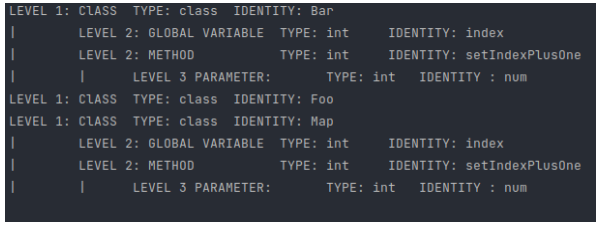
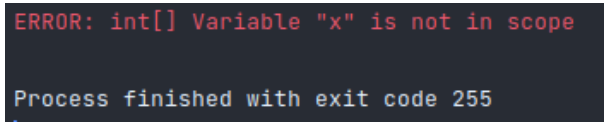
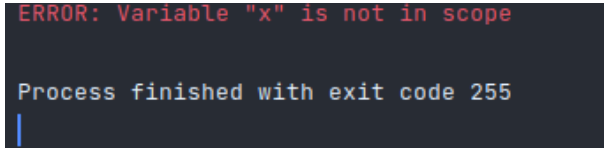
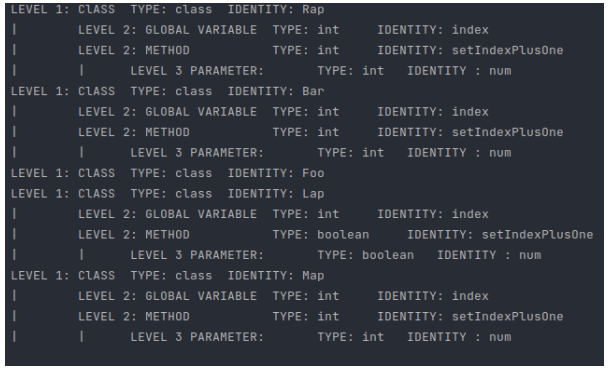
3 Testing

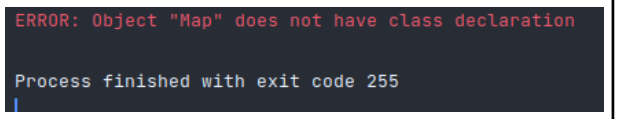
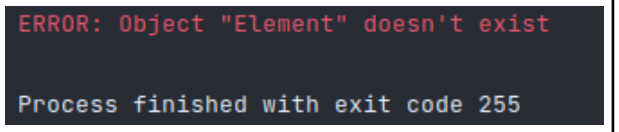
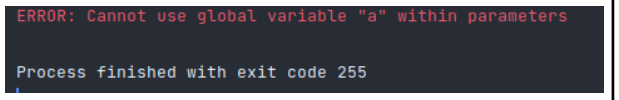
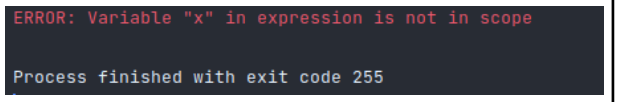
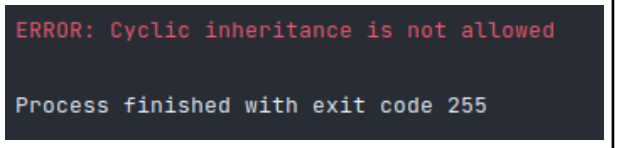
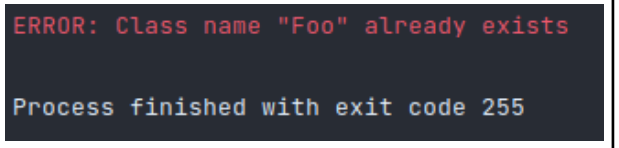
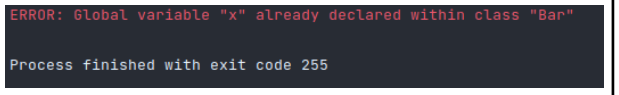
For this practical, I decided that having a test-driven approach would be quite beneficial given the large amount of edge cases that appear for this practical. Hence I decided to make a list of tests to try along with what should be expected for each edge case. I then decided to use very simple base case tests to ensure that each small edge case is accounted for. I decided to split the tests into two groups: scope checking and type checking. Currently I have a total of 16 MiniJava tests for scope checking and a total of 17 MiniJava tests for type checking. The list of tests can be found at the bottom of the report detailing the file, description of test, expected value and actual value in the form of screenshots. The source code of these tests can be found under the P1 working directory: ./P1/tests/myTests/. Apart from my custom tests, my implementation can run the given tests successfully and as intended.

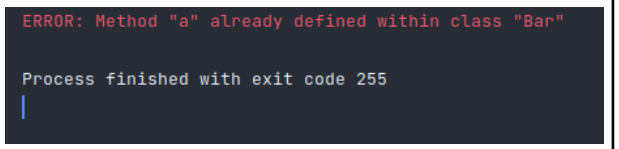
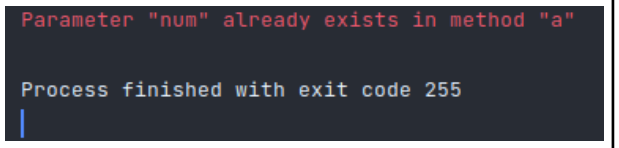
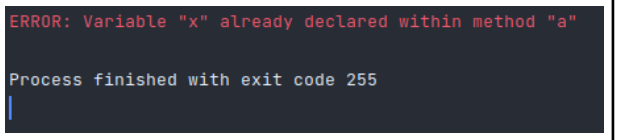
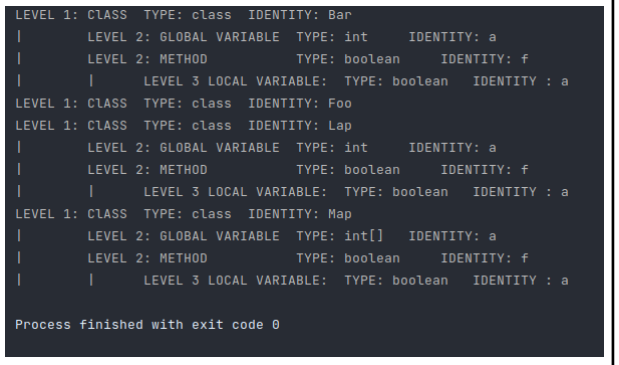
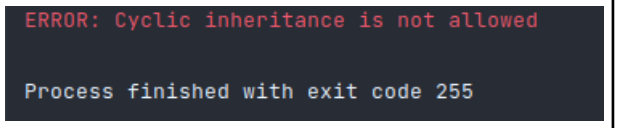
4 Evaluation

This practical deepened my understanding and knowledge of the semantic analysis process of a compiler. Through this practical, I was challenged to think of many considerations one would have when implementing this, such as variable shadowing, inheritance, overloading, overriding, type checking through stacks, etc. Though my implementation has been well-tested and documented, there are still a few things I would add given more time. One thing I would consider is the speed and efficiency of my implementation. While using hashmap to store the records proved to be useful for fast lookups, there are still possible improvements for the speed of implementation. One such example is through my inheritance pass. For each class, I look through its parents and the parents of its parents with a new ArrayList each time. If I kept a better record of the relations between each of the classes, I could speed up this check by not having to re-add every superclass' global variables and methods, but just the parent above. As an extension, I would try to combine declaration and initialization in the same line.

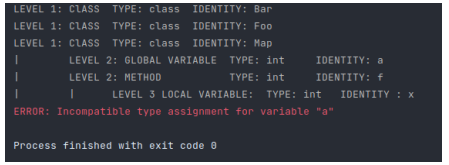
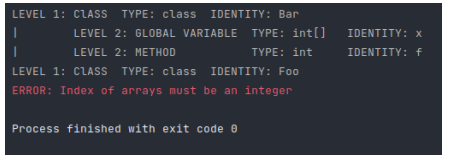
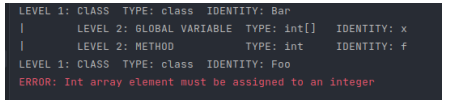
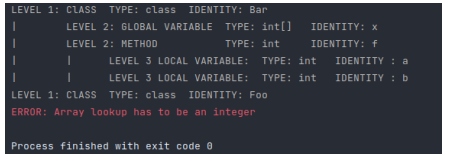
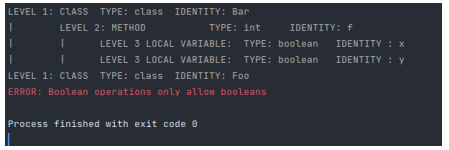
Scope Checking Tests

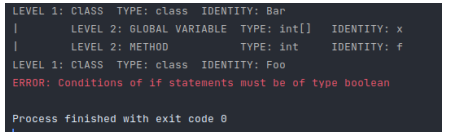
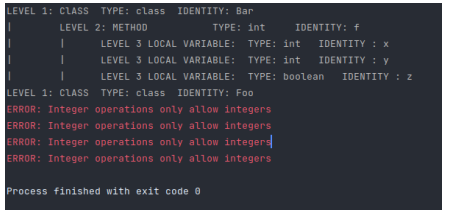
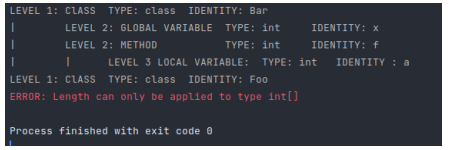
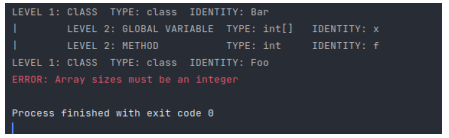
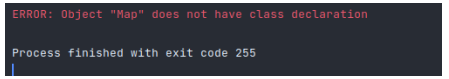
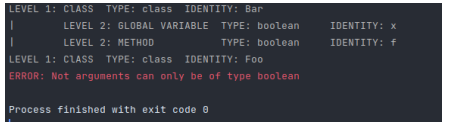
File Name	Description	Expected	Actual
AccessToSuperGlobalVarsAndMethods.java	Checking that child class has access to superclass's global variables and methods	Symbol table populated with superclass global variables and methods	
AssigningArrayNotInScope.java	Checking an int[] array not in scope	ERROR: int[] Variable "x" is not in scope	
AssigningVarNotInScope.java	Checking a variable not in scope	ERROR: Variable "x" is not in scope	
ExtendedAccessToSuperGlobalVarsAndMethods.java	Checking that child class has access to all the superclasses of its superclass's global variables and methods	Symbol table populated with extended superclass global variables and methods	

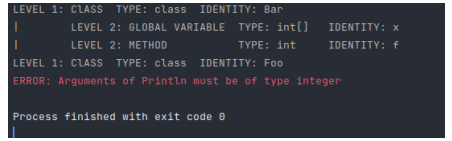
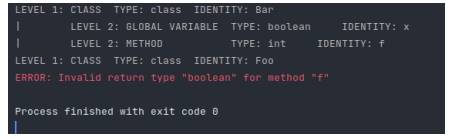
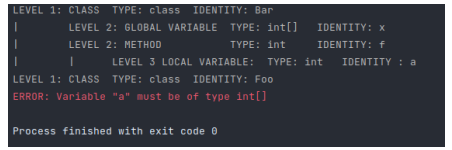
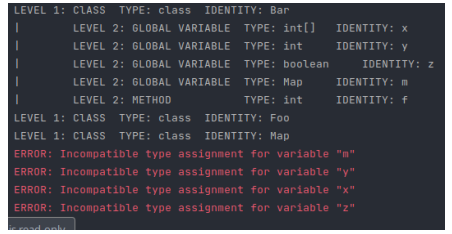
InvalidObjectDeclWithNoExistingClass.java	Checking assigning a variable to an object with no class declaration	ERROR: Object "Map" does not have class declaration	
InvalidObjectVarDeclaration.java	Checking declaring a var to an object with no class declaration	ERROR: Object "Element" doesn't exist	
InvalidUseGlobalVarInParam.java	Checking using the same name as a global variable in a formal parameter list	ERROR: Cannot use global variable "a" within parameters	
InvalidVarExprScope.java	Checking assigning a variable to an expression that is not in scope	ERROR: Variable "x" in expression is not in scope	
LongCyclicInheritance.java	Checking for long cycles of inheritance	ERROR: Cyclic inheritance is not allowed	
RepeatedClassNames.java	Checking for repeated class declarations	ERROR: Class name "Foo" already exists	
RepeatedGlobalDeclarationInClass.java	Checking for repeated global variables declarations in the same class	ERROR: Global variable "x" already declared within class "Bar"	

RepeatedMethodDeclarationInClass.java	Checking for repeated method declarations in the same class (No overloading)	ERROR: Method "a" already defined within class "Bar"	
RepeatedParameterInMethod.java	Checking for repeated parameter names within a formal list	Parameter "num" already exists in method "a"	
RepeatedVarDeclarationInMethod.java	Checking for repeated variable declarations in a method	ERROR: Variable "x" already declared within method "a"	
ShadowingVars.java	Checking for variable shadowing between class scopes (including inherited) and method scopes	Valid symbol table with shadowing between classes Bar and Map, class and method Bar and f	
ShortCyclicInheritance.java	Checking for short cycles of inheritance	ERROR: Cyclic inheritance is not allowed	

Type Checking Tests

File Name	Description	Expected	Actual
ContinuesAfterError.java	Checking that program continues as normal if type error is found (no cascading errors)	ERROR: Incompatible type assignment for variable "a" → printed after symbol table	
InvalidArrAssignIndexExpr.java	Checking non-int array indexes in array expressions	ERROR: Index of arrays must be an integer	
InvalidArrAssignValueExpr.java	Checking non-int assigned values for array indexes	ERROR: Int array element must be assigned to an integer	
InvalidArrIndexLookupExpr.java	Checking non int[] variables using a array lookup	ERROR: Array lookup has to be an integer	
InvalidBoolOperation.java	Checking boolean operations on an integer and a boolean	ERROR: Boolean operations only allow booleans	

InvalidIfStatement.java	Checking non-boolean conditions in “if” statements	ERROR: Conditions of if statements must be of type boolean	
InvalidIntOperation.java	Checking int operations on booleans	ERROR: Integer operations only allow integers (+) ERROR: Integer operations only allow integers (*) ERROR: Integer operations only allow integers (<) ERROR: Integer operations only allow integers (-)	
InvalidLengthExpr.java	Checking non-int[] variables calling the length method	ERROR: Length can only be applied to type int[]	
InvalidNewArrExpr.java	Checking non-int lengths given in new int[] statements	ERROR: Array sizes must be an integer	
InvalidNewObjExpr.java	Checking for new object initialization that don’t have a corresponding class	ERROR: Object "Map" does not have class declaration	
InvalidNotExpr.java	Checking non-boolean variables using the “not” expression	ERROR: Not arguments can only be of type boolean	

InvalidPrintlnStatement.java	Checking non-int variables using the “Println” expression	ERROR: Arguments of Println must be of type integer	
InvalidReturnExpr.java	Checking non-matching return types with method types	ERROR: Invalid return type "boolean" for method "f"	
InvalidVarArrAssignExpr.java	Check for non-int[] variables using array assignments	ERROR: Variable "a" must be of type int[]	
InvalidVarAssignExpr.java	Checking for non-matching types during variable assigning	ERROR: Incompatible type assignment for variable "m" ERROR: Incompatible type assignment for variable "y" ERROR: Incompatible type assignment for variable "x" ERROR: Incompatible type assignment for variable "z"	
InvalidWhileStatement.java	Checking for non-boolean expressions in while statements	ERROR: Conditions of while statements must be of type boolean