Introduction

The purpose of this assignment is to create a program that can run semantic analysis and generate three address code from a CCAL source file.

The main program is located in the ccal.java file. It first generates a lexer and a parser from the ccal.g4 grammar file before using the parser to generate a parse tree which is later used for semantic analysis and the generation of three address code.

Semantic Analysis

Syntax errors are caught by the parser. However, there are additional checks related to the meaning of statements and expressions which are not. The purpose of semantic analysis is to check for errors involving type, scope and other important checks which cannot be done by the parser.

In my implementation, semantic analysis is performed by an instance of a class named SemanticAnalyzer which inherits from the CCAL base visitor in order to have the ability to traverse the parse tree.

My implementation of the SemanticAnalyzer class can perform the following semantic checks and outputs an error if any of these checks fail:

- Every identifier is declared in scope before use.
- No duplicate identifiers in each scope.
- The left-hand side and right-hand side of a variable assignment both have the same type (e.g. const x:integer = 5).
- All invoked identifiers have a declared function associated with them.
- All functions are invoked.
- All invoked functions are invoked with the correct number of arguments.
- All parameters of invoked functions have the correct types.
- The type of an expression returned from a function should be the same as the declared return type of that function.
- All variables are written to and read at least once.
- The main function is defined.

Symbol Table

The symbol table is an essential component of the SemanticAnalyzer class. It is implemented in its own SymbolTable class and contains the following sub-tables. Part of my symbol table implementation is based on the description of the symbol table in the lecture notes with additional tables to assist semantic analysis.

```
// key: scopeName, value: linked list of variables in that scope
private final HashMap<String, LinkedList<String>> scopesTable;

// key: identifierName-scopeName,
value(string array):
[hasValue, type, qualifier, numTimesWritten, numTimesRead]
private final HashMap<String, String[]> identifierTable;

// needed to compare the types of return expressions and function return
types
// key: expression,
value(string array): [expression, identifierName, numChildrenOfExpression]
private final ArrayList<String[]> ExpressionStack;

// key: functionName,
value(string array):
[numArgs, numTimesCalled, argType1, argType2, ..., argTypeN]
private final HashMap<String, String[]> functionInfoTable;
```

Handling Scope

The scopes of identifiers are recorded by scopesTable which is a hash map where each key is a scope and each key is associated with a linked list value containing the names of identifiers in that scope. Global variables and functions are stored in the global scope and local variables are stored in a scope with the same name as their parent function.

The current scope is stored in a class variable in the SemanticAnalyzer class named currentScope and is initialized as “global” to represent the global scope. Every time a new function is visited, the current scope is changed to the name of that function.

Checking for Duplicates

When a new variable declaration is visited, the variable is first searched for in the scopes table to check if it is a duplicate. The current scope is used as a key to retrieve all variables in the current scope from scopesTable. If the new variable is found in the linked list, it must be a duplicate and an error is printed. Otherwise, the new variable is added to the scopesTable.

New variables are also added to a hashmap named identifierTable which records more information about the variable such its qualifier (eg. var, const) and and the number of times the variable has been read and written.

When a scope is exited, all variables in that scope are deleted by deleting the key and value associated with that scope in the scopesTable. The variable is not deleted from the identifierTable as it is needed to perform global semantic checks after the entire tree has been walked.

Checking for Undefined Variables

Identifiers that are variables are read in the following situations:

- If they are on the right-hand side of assignment statements.
- If they are passed as arguments to functions.
- If they occur in conditions.

When an identifier is read, we first need to check that it has already been defined. This can be done by using the scopesTable to find the variables associated with a scope and then searching for the current variable. If it is not found in scopesTable, it was used before declaration and an error will be printed.

Type Checking

Type checking is done in the following situations:

- In assignment statements, we need to check that the identifier on the left-hand side and the expression on the right-hand side have the same type.
- Arguments passed to functions need to be of the correct type.
- Values returned by functions should be of the same type as the function return type.

A statement with four children is an assignment statement and when it is visited we calculate the types of the left and right-hand side before comparing them. The type of the variable can be looked up in the identifiersTable. Finding the type of the expression is more complex. I wrote a method named findExpressionType that breaks expressions into variables and numbers. Number literals are given the type “integer” and variables in the expression are looked up in the identifierTable for their type. If a variable is undeclared, it will have the type “NA”. In addition or subtraction expressions, we need to check that every identifier and number in the expression has the “integer” type or an error will occur. If every identifier and number has the “integer” type, the expression will be given the type “integer” and can be compared with the type of the variable on the left-hand side of the assignment statement. If the variable being assigned and the expression don’t have the same type, an error will occur. For example, a boolean variable assigned to the expression “2 + 3” would cause an error.

We can also check the type of arguments passed into a function. Every time a new function is encountered, it is added to the functionInfoTable which records its number of arguments and their types. When the function is invoked, a loop iterates through the function’s arguments and compares them to the expected argument types in the functionInfo table. Any mismatch between the actual type and expected type causes an error.

The return type of functions can be checked with the expression stack named expressionStack in the SymbolTable class. When a function is encountered, each expression is added to the expressionStack and the final expression will always be the return expression. When the next function is reached, the type of the return expression in the previous function can be checked by popping the expression off the expression stack, and comparing its type to the type of its parent function. Once the type has been checked, the expression stack is emptied and the process begins again in the next function.

Checking Function Calls

A hashmap named functionInfo stores important information about functions including their number of arguments, the number of times they have been called and the types of all their arguments.

Every time a function is invoked, in addition to checking the types of the arguments, the number of arguments passed to the function is checked and an error is printed if too many or too few arguments were used. Each time a function is invoked, a counter that tracks how many times it was called is incremented.

At the end, when the entire tree has been visited, an error will occur if any function has been invoked zero times as this means it has not been invoked.

Global Semantic Checks

There are several global flags defined in the IntermediateCodeGenerator class which keep track of the global state of several errors. If a local error occurs while analyzing the parse tree, a global semantic flag related to that error will be updated. For example, any type error sets the noTypeErrors global flag to false. In order for the semantic analysis to pass, all global flags must be set correctly. The final global check is done by the allSemanticChecksPassed method which returns false if any of the global flags are incorrectly set and true otherwise. If the method returns false, an error will be printed saying that the semantic analysis phase failed and the program will terminate. If it returns true, the program continues to the three address code generation phase.

Three Address Code Generation

The next phase of the program is to convert the parse tree into three address code. Three address code is a low level programming language which is similar to assembly language. One of the main differences is that three address code is machine independent whereas assembly code is not.

If any semantic errors occur, the program will terminate before generating code. This behaviour is desired as semantically incorrect code is likely to generate intermediate code that is also incorrect.

If all semantic checks pass, an instance of a class named IntermediateCodeGenerator converts the CCAL input from input.ccl into three address code and outputs the resulting code to another file named output.ir. Like the class that performs semantic analysis, IntermediateCodeGenerator also inherits from the CCAL base visitor so that it can walk the parse tree.

Assignment Statements

My implementation of the three address code generator ignores variable declarations and only prints assignment statements. Simple assignment statements such as “x = 1” are easy to convert into three address code. The challenge is converting more complex statements such as “x = 1 + 2 + 3” into three address code as there must be no more than one operator per statement. Thus complex statements like the one above must be converted into multiple statements involving temporary variables. For example, the statement “x = 1 + 2 + 3” would be converted into the following three address code:

```
_t1 = 1 + 2
_t2 = _t1 + 3
x = _t2
```

The necessary temporaries can be generated by printing out the contents of the first expression node before recursively visiting each subexpression node and generating a statement for each node. When all children have been visited and all temporaries have been generated, the identifier is assigned to the value of the last temporary.

Function Calls

In my implementation function names are printed as upper case labels and functions are called by listing the parameters used by the function and then calling the function as shown below:

```
ADD:
  _t1 = a + b
  _t2 = _t1 + c
  result = _t2
  return

MAIN:
  a = 1
  b = 2
  c = 3
  param a
  param b
  param c
  answer = call add, 3
```

If Statements

Like the expressions in assignment statements, conditions also must have only one boolean operator per line and conditions with multiple operators must be broken up into multiple conditions as shown below. This is done in a similar way to the assignment statements except condition trees are being recursively traversed instead of an expression tree. Another difference is that the variable temporaries involve a top-down traversal whereas the condition temporaries use a bottom-up tree traversal.

```
MAIN:
  a = 1
  b = 2
  c = 3
  _condition1 = a < b
  _condition2 = b < c
  _condition3 = _condition1 && _condition2
  if _condition3 goto _IF1
  goto _ELSE1
_IF1:
  a = a + 1
  goto _END_IF1
_ELSE1:
  b = b + 1
_END_IF1:
```

To generate if statements and while loops I used and a global ifState and whileState integer variable to inform the visitor on how to visit nodes. For example, when the ifState is equal to 1, this means the initial conditions are being printed and no statements should be printed. Using the ifState and whileState variables therefore ensures that code is printed correctly, that there are no duplicate statements and that the code is printed in the right order.

While Loops

Similar to if statements, while loops can be created in three address code using a combination of labels and conditions. I implemented while loops by having an unconditional jump statement at the bottom of the loop and a break statement at the top:

```
MAIN:
  i = 0
  _condition1 = i < 10
_WHILE1:
  ifFalse _condition1 goto _END_WHILE1
  i = i + 1
  goto _WHILE1
_END_WHILE1:
```

Limitations

While my three address code generator does handle most situations it does have limitations. The main limitation is that it does not have the ability to convert nested if or while loops into three address code.

Sources

While working on the three address code generator I used the following information sources:

- The Compiler Construction module slides.
- These slides: https://stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/13/Slides13.pdf
- Stackoverflow, Baeldung and other sites for information on the Java programming language.

Unit Tests

I also wrote unit tests for the semantic analyzer and three address code generator which can be found in TestSemanticAnalyzer.java and TestCodeGeneration.java. The unit tests are written using the Junit5 testing framework and their purpose is to check that all aspects of semantic analysis and three address code generation work as expected.