

Assignment 2: Semantic Analysis and Intermediate Representation



Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

Name: *Fíonn Hourican, 21477216*

School of Computing & Engineering

CSC1098 Compiler Construction

Instructor: *Dr David Sinclair*

Due Date *02/12/2004*

Abstract

This assignment aims to enhance the lexical and syntax analyser from Assignment 1 by adding semantic analysis and generating an intermediate representation in the form of 3-address code. Essential tasks include creating a parse tree, implementing a scope-aware symbol table, and conducting essential semantic checks. The generated 3-address code will serve as an intermediate representation, simplifying further optimisation and code generation processes in the compilation pipeline.

Plagiarism Declaration

I understand that the University regards breaches of academic integrity and plagiarism as grave and serious.

I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy.

I have identified and included the sources of all the facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations, paraphrasing, and discussion of ideas from books, journal articles, internet sources, module text, or any other source are acknowledged, and the sources cited are identified in the assignment references.

I declare that this material, which I now submit for assessment, is entirely my work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

I have used the DCU library referencing guidelines and/or the appropriate referencing system recommended in the assignment guidelines and/or programme documentation.

Signed: Fionn Hourican

Assignment 2: Semantic Analysis and Intermediate Representation

I found this Assignment extremely challenging, especially compared to the previous Assignment. I also found it extremely helpful in understanding more about Symbol Tables, Semantic checks, and intermediate representation, especially in three address codes. Due to time constraints, I did not get the assignment fully completed, and given more time, I would have developed more Semantic Checks and finished generating intermediate code representation.

Generating The Parse Tree

Antler automatically generated the Parse Tree by parsing the input from a .cal file. I extended this in my Eval visitor to Print the tree in a more readable format to ensure that the tree was generated as expected. On the right is how my Parse tree looked for a basic cal file with nothing declared.

```
prog
|- decl_list
|- function_list
|- main
|-   |- main
|-   |- begin
|-   |- decl_list
|-   |- statement_block
|-   |- end
```

Visualising the parse tree generated by ANTLR for the .cal file helped me understand the tree's structure and ensured that the grammar rules I defined were correctly applied. I verified the grammar's implementation by extending the Eval visitor to print the tree in a more readable format. I debugged any issues by identifying where the parsing process might not align with expectations. This also provided insight into how the input was parsed into tokens and grouped according to the grammar's structure, clarifying how the language was interpreted. This readable representation of the parse tree was invaluable in understanding the hierarchical relationships between nodes, which was crucial for developing semantic analysis and intermediate representation generation, which I will talk about later.

Symbol Table that can handle scope

The next part of the assignment involved implementing a Symbol Table capable of handling scopes, which I achieved using a SymbolTable class. The implementation was based on the description in the notes, while I also researched how to implement this online and found different techniques described here [2]. The SymbolTable class manages symbols and scopes by employing a stack to track the current scope and a list to store all defined scopes. It also uses maps to handle function parameter counts, return types, and written and read variables.

To handle scopes, I implemented methods such as enterScope to push a new map onto the stack for a new scope and exitScope to remove the current scope. Symbols like variables, constants, and parameters are added to the current scope using methods like insert, insertVariable, insertConstant, and insertParameter. These symbols are stored with their types, ensuring each scope is appropriately encapsulated. For functions, the insertFunctionParameterCount and insertFunctionReturnType methods record parameter counts and return types, respectively.

Symbol lookups were implemented with the lookup and getType methods, which search from the innermost scope outward to ensure correct symbol resolution. To track variable usage, I added methods like markVariableWritten and markVariableRead, which mark variables as written to or read from. Their respective sets can be retrieved with getWrittenVariables and getReadVariables.

Symbol Table			
Scope: global			
ID	Type	Memory Location	Return Type
i	var	3	int
test_fn	function	N/A	int
Scope: function test_fn			
ID	Type	Memory Location	Return Type
x	param	N/A	int
i	var	3	int
Scope: main			
ID	Type	Memory Location	Return Type
i	var	3	int

The Scope class, nested within the SymbolTable, represents individual scopes by storing their name and a map of their symbols. This modular design allowed the symbol table to be seamlessly integrated into the visitor classes, such as FunctionVisitor for handling function declarations, DeclVisitor for

variable and constant declarations, StatementVisitor for statement processing, and ExpressionVisitor for tracking variable reads and writes. I also implemented debugging methods like printCurrentScope and printAllScopes, which display symbols and their attributes for validation and memory management. The Symbol Table effectively managed symbol resolution and scope handling through this implementation, ensuring robust program analysis.

Semantic checks

The next step was to add Semantic checks to the compilation process. Many of these checks were described in the assignment description, but I added a few more to my compiler. If I was to repeat this step I would of made a separate class defining each check and then call the checks in each necessary visitor. This would of made the code more modular and easy to understand.

To ensure that every identifier is declared within scope before being used, the lookup method of the SymbolTable is called whenever an identifier is encountered, such as in an assignment or expression. This method checks if the identifier has been declared in the current or

any enclosing scope. If the identifier is not found, an error message is printed, preventing the use of undeclared variables and ensuring proper scope management.

```
@Override
public Object visitAssignmentStatement(calParser.AssignmentStatementContext ctx) {
    String id = ctx.ID().getText();
    // Check if variable is declared
    if (symbolTable.lookup(id) == null) {
        System.err.println("Error: Variable " + id + " not declared.");
        return null;
    }
}
```

The next check ensured no identifier was declared more than once in the same scope.

This was implemented using the insert method of the SymbolTable, which adds an identifier to the current scope. Before insertion, the method checks if the identifier exists within the current scope. If it does, an error message is printed, preventing duplicate declarations and maintaining the integrity of scope definitions.

```
public void insert(String name, String type) {
    if (!scopes.isEmpty() && scopes.peek().containsKey(name)) {
        System.err.println("Error: Variable " + name + " is already declared in the current scope.");
    } else {
        scopes.peek().put(name, type);
    }
}
```

The program also checks whether the left-hand side of an assignment is a variable of the correct type. When an assignment statement is visited, the type of the left-hand side variable is retrieved from the symbol table, and the type of the right-hand side expression is evaluated. An error message is printed to prevent type mismatches if the types do not match.

```
@Override
public Object visitAssignmentStatement(calParser.AssignmentStatementContext ctx) {
    String id = ctx.ID().getText();
    String expectedType = symbolTable.lookup(id);
    String actualType = expressionVisitor.getType(ctx.expression());
    if (!expectedType.equals(actualType)) {
```

```
        System.err.println("Error: Type mismatch for variable " + id + ". Expected  
" + expectedType + " but got " + actualType + ".");  
    }  
}
```

Arithmetic operations are validated to ensure that their operands are integer variables or constants. During the visit to an arithmetic operation, the types of all operands are evaluated. If any operand is not an integer, an error message is printed.

```
@Override  
public Object visitPlus(calParser.PlusContext ctx) {  
    String leftType = expressionVisitor.getType(ctx.expression(0));  
    String rightType = expressionVisitor.getType(ctx.expression(1));  
    if (!leftType.equals("int") || !rightType.equals("int")) {  
        System.err.println("Error: Both operands of an arithmetic  
operator must be integers.");  
    }  
}
```

The program checks that all operands are either boolean variables or constants for boolean operations. If any operand is not a boolean, an error message is issued to prevent invalid operations.

```
@Override  
public Object visitAnd(calParser.AndContext ctx) {  
    String leftType = expressionVisitor.getType(ctx.condition(0));  
    String rightType = expressionVisitor.getType(ctx.condition(1));  
    if (!leftType.equals("bool") || !rightType.equals("bool")) {  
        System.err.println("Error: Both operands of a boolean operator must be  
booleans.");  
    }  
}
```

It also ensures that a function is declared for every invoked identifier. When a function call is visited, the symbol table's lookup method verifies whether the function has been declared. If the function is not found, an error message is printed to ensure that only declared functions are called.

```
@Override
```

```

public Object visitFunctionCall(calParser.FunctionCallContext ctx) {
    String functionName = ctx.ID().getText();
    if (symbolTable.lookup(functionName) == null) {
        System.err.println("Error: Function " + functionName + " not declared.");
        return null;
    }
}

```

The number of arguments in a function call is compared to the expected number of parameters stored in the symbol table. An error message is printed if the number of arguments does not match.

```

@Override
public Object visitFunctionCall(calParser.FunctionCallContext ctx) {
    String functionName = ctx.ID().getText();
    int expectedParamCount = symbolTable.getFunctionParameterCount(functionName);
    int actualParamCount = ctx.arg_list().expression().size();
    if (expectedParamCount != actualParamCount) {
        System.err.println("Error: Function " + functionName + " expects " +
            expectedParamCount + " arguments but got " + actualParamCount + ".");
    }
}

```

Variables are analysed to ensure they are both written to and read from. During program evaluation, variables written to are added to a writtenVariables set, and variables read from are added to a readVariables set in the symbol table. After evaluation, these sets are compared. Warnings are issued for variables written to but never read or read but never written to.

```

public void markVariableWritten(String name) {
    writtenVariables.add(name);
}

public void markVariableRead(String name) {
    readVariables.add(name);
}

public void checkVariableUsage() {
    for (String var : writtenVariables) {
        if (!readVariables.contains(var)) {
            System.err.println("Warning: Variable " + var + " is written to but
never read.");
        }
    }
}

```



```
    }  
    for (String var : readVariables) {  
        if (!writtenVariables.contains(var)) {  
            System.err.println("Warning: Variable " + var + " is read from but  
never written to.");  
        }  
    }  
}
```

Functions are tracked to ensure every function declared is called. Functions that are invoked are added to a set during program evaluation. This set is then compared to the set of declared functions. Warnings are generated for functions that are declared but never called.

```
public void markFunctionCalled(String name) {  
    calledFunctions.add(name);  
}  
  
public void checkFunctionUsage() {  
    for (String function : declaredFunctions) {  
        if (!calledFunctions.contains(function)) {  
            System.err.println("Warning: Function " + function + " is declared but  
never called.");  
        }  
    }  
}
```

Additional Semantic Checks

Ensuring that functions return values of the correct type is essential for maintaining type safety and correctness in a program. If a function returns a value of an unexpected type, it can cause runtime errors, incorrect behaviour, and debugging challenges. This semantic check identifies such issues early in development, ensuring functions adhere to their declared return types. When a function is declared, its return type is stored in the symbol table using the `insertFunctionReturnType` method. When a return statement is encountered, the type of the returned value is evaluated and compared against the expected return type stored in the symbol

table. If a mismatch is detected, an error message is printed, highlighting the discrepancy and maintaining the integrity of type constraints.

```
public class FunctionVisitor extends calBaseVisitor<Object> {
    private SymbolTable symbolTable;
    private List<String> tacInstructions;

    public FunctionVisitor(SymbolTable symbolTable, List<String> tacInstructions) {
        this.symbolTable = symbolTable;
        this.tacInstructions = tacInstructions;
    }

    @Override
    public Object visitFunction(calParser.FunctionContext ctx) {
        String functionName = ctx.ID().getText();
        String returnType = ctx.type().getText();
        symbolTable.insertFunctionReturnType(functionName, returnType);

        return null;
    }
}
```

```
@Override
public Object visitReturnStatement(calParser.ReturnStatementContext ctx) {
    String functionName = currentFunctionName; // Assume this is set when entering
    the function
    String expectedReturnType = symbolTable.getFunctionReturnType(functionName);

    // Evaluate the type of the returned value
    String actualReturnType = expressionVisitor.getType(ctx.expression());

    // Compare the actual return type with the expected return type
    if (!expectedReturnType.equals(actualReturnType)) {
        System.err.println("Error: Function " + functionName + " expected return
type " + expectedReturnType + " but got " + actualReturnType + ".");
    }
    return null;
}
```

The symbol table is checked whenever a function is declared to avoid duplicate function declarations to ensure the function name is not already in use. If a duplicate declaration is found, an error message is raised.

```
String functionName = ctx.ID().getText();
    if (symbolTable.isDeclaredInCurrentScope(functionName)) {
        System.err.println("Error: Function " + functionName + " is already
declared in the current scope. here");
    }
```

Lastly, the program ensures that constant declarations are not redeclared later in the program. If this were allowed, variables and constants would be the same.

```
if (symbolTable.getType(id).equals("const")) {
    System.err.println("Semantic error: Constant " + id + " cannot be
reassigned.");
}
```

By implementing these checks, the program enforces strict adherence to semantic rules, ensuring correctness and reliability while preventing common errors. Implementing some of these checks was made simple due to my Symbol Table class, which helped me realise how important and useful the symbol table is.

Intermediate Representation

The Final stage of the assignment was to generate an Intermediate Representation using a 3-address code, particularly code that could be interpreted using the TACi.jar file included in the submission. Before I started this section, I read this article [1] and looked at the TACi.jar description. These were the basis for implementing my 3-address code. I created each instruction and added them to a tacInstructions Array.

The Three Address Code (TAC) instructions are generated during the visitation of the parse tree by various visitor classes. They represent the program's operations and control flow in a simplified, intermediate form. The intermediate follows the form described in the jar file mentioned above.

When an assignment statement is encountered, the `visitAssignmentStatement` method in the `StatementVisitor` class is invoked. Within this method, the variable name and the value to be

```
String id = ctx.ID().getText();
String value = ctx.expression().getText();
tacInstructions.add(id + " = " + value);
```

assigned are retrieved. Three-address code (TAC) instructions are generated to represent this operation and added to the list of

instructions, effectively capturing the assignment operation.

When an arithmetic operation is encountered, the corresponding method in the `ExpressionVisitor` class is called. Temporary variables are generated to store the operation. Originally, I assigned each side of the equation a temporary value and constructed the operation using these variables. For example, $X = Y + Z$ was added as $t1 = Y$, $t2 = Z$, and $t3 = t1 + t2$. However, I realised this was unnecessary as, based on the language description, arithmetic could only be done in a 3-address code format.

```
// Get the left and right operands
calParser.FragContext leftOperand = ctx.frag(0);
calParser.FragContext rightOperand = ctx.frag(1);

String leftValue = getValue(leftOperand);
String rightValue = getValue(rightOperand);

String tempVar = newTempVar();
String op = ctx.binary_arith_op() instanceof
calParser.PlusContext ? "+" : "-";

// Add to Tac instructions
tacInstructions.add(tempVar + " = " + leftValue + " " + op +
" " + rightValue);
```

When a function is declared, a Tac instruction is created to mark the scope of a function. The function visitor then visits its children, adding all assignment statements to the `tacInstructions` array, declarations were not added as these are unnecessary.

The TAC instructions are generated for all control flow statements in all scopes. This includes handling both while loops and if-else statements. For if-else statements, TAC instructions are generated to evaluate the condition and direct the control flow appropriately. Instructions are added to jump to the appropriate loop labels based on the condition, execute the if block, and optionally handle the else block if it exists. Loop labels manage the flow between these blocks and ensure a seamless transition to subsequent code.

```
// Add TAC instructions for the if condition
tacInstructions.add("if " + condition + " goto " + ifLabel);
tacInstructions.add("goto " + elseLabel);

// Add TAC instructions for the if block
tacInstructions.add(ifLabel + ":");
visit(ctx.statement_block(0));
tacInstructions.add("goto " + endLabel);

// Add TAC instructions for the else block (if it exists)
tacInstructions.add(elseLabel + ":");
if (ctx.statement_block().size() > 1) {
    visit(ctx.statement_block(1));
}
tacInstructions.add(endLabel + ":");
```

TAC instructions are generated for while loops to evaluate the loop's condition and direct the control flow accordingly. Instructions are added to jump to the loop body if the condition is true or to exit the loop otherwise. The loop's body is visited recursively, and a jump back to the start of the loop ensures that the condition is re-evaluated. Loop labels manage the flow between the beginning, body, and end of the loop, creating a structured representation of the

loop's logic in the intermediate code. These control flow instructions ensure that all branches and iterations are accurately captured in the TAC.

```
// Add TAC instructions for the while condition
tacInstructions.add("if " + conditionTemp + " goto " + bodyLabel);
tacInstructions.add("goto " + endLabel);

// Add TAC instructions for the body of the loop
tacInstructions.add(bodyLabel + ":");
visit(ctx.statement_block());
tacInstructions.add("goto " + startLabel);

// Add TAC instructions for the end of the loop
tacInstructions.add(endLabel + ":");
```

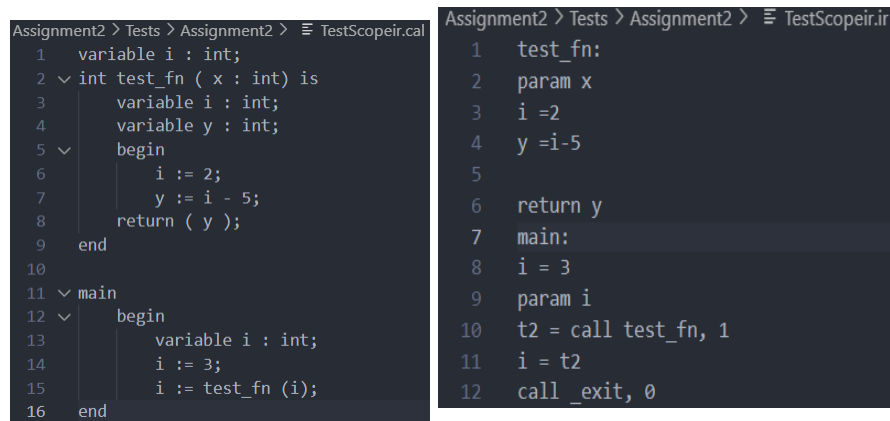
Function calls also had to be transformed into TAC instructions. When a function is called, it is added to the array in the following format.

```
tacInstructions.add(tempvar + " = " + "call " + id + ", " +
parameters.size());
```

These TAC instructions were then output to a file with the same name as the input file with .cal replaced by .ir.

Example Execution

I created a file called TestScopeir.cal, which implements a simple function call. The .cal file on the left makes the .ir file on the right when the command “*java cal.java* ..\Tests\Assignment2\TestScopeir.cal”



```
Assignment2 > Tests > Assignment2 > TestScopeir.cal
1  variable i : int;
2  int test_fn ( x : int) is
3      variable i : int;
4      variable y : int;
5      begin
6          i := 2;
7          y := i - 5;
8          return ( y );
9      end
10
11 main
12 begin
13     variable i : int;
14     i := 3;
15     i := test_fn (i);
16 end

Assignment2 > Tests > Assignment2 > TestScopeir.ir
1  test_fn:
2  param x
3  i =2
4  y =i-5
5
6  return y
7  main:
8  i = 3
9  param i
10 t2 = call test_fn, 1
11 i = t2
12 call _exit, 0
```

This follows the format specified for TACi.jar and can be interpreted by running “*java -jar* ..\TACi.jar

..

```
Debug - Printing internal tables
Values:
i: -3
y: -3
t2: -3

Entry Points:
test_fn
main

Pointers:
```

Here, we can see the final values, which evaluate i as -3. It also shows each entry point. This also works for control flow statements, as seen in the Appendix, which follow the same steps.

Reflection

This Assignment was extremely challenging, and I was unable to complete it due to time constraints. The TAC is implemented for basic arithmetic, function calls, and control flow statements. However, it does not implement full boolean operations and conditional solutions. Given more time, I would have implemented this solution.

My implementation of the TAC instruction is also not optimal, as it creates some unnecessary temporary variables. I know this is the next stage in a compiler, but this could have been avoided due to the language's definition, which restricts arithmetic to 3-address code.

Regarding my program structure, I tried to separate Visitors into separate classes to help make the code more modular. If I were to start again, one area for improvement would be the organisation and modularisation of the semantic checks and code generation processes. Specifically, I would have separated the semantic checks into distinct visitor classes dedicated to different checks, such as type checking, scope resolution, and usage analysis. This would

enhance the code's readability and maintainability by clearly delineating each visitor's responsibilities.

For the intermediate representation (IR) generation, I would have created a dedicated IR generation module that abstracts the details of TAC (Three Address Code) generation, allowing for more straightforward modifications and extensions in the future. This module would interact with the symbol table and the semantic analysis results to produce a clean and well-structured IR.

Running Test file

- `java cal.java .\Tests\TestScopeir.cal`
- `java -jar ..\TACi.jar .\Tests\TestScopeir.ir`

Appendix

```

variable i : int;
int test_fn ( x : int) is
  variable i : int;
  variable y : int;
  begin
    i := 1;
    if (i > 0) begin
      i := i - 2;
      y := i - 5;
    end
    else begin
      i := i + 2;
      y := i + 5;
    end
    return ( y );
  end
main
  begin
    variable i : int;
    i := 3;
    i := test_fn (i);
  end

```

```

test_fn:
param x
i = 1
if i>0 goto L1
goto L2
L1:
t1 = i - 2
i = t1
t2 = i - 5
y = t2
goto L3
L2:
t3 = i + 2
i = t3
t4 = i + 5
y = t4
L3:
t5 = i - 2
i = t5
t6 = i - 5
y = t6
t7 = i + 2
i = t7
t8 = i + 5
y = t8
return y
main:
i = 3
param i
t2 = call test_fn, 1
i = t2
call _exit, 0

```

```

Debug - Printing internal tables
Values:
t5: -3
t6: -8
t7: -1
t8: 4
i: 4
y: 4
t1: -1
t2: 4

Entry Points:
test_fn
L1
L2
t2: 4

Entry Points:
test_fn
L1
L2

Entry Points:
test_fn
L1
L2

Entry Points:
test_fn
L1
L2
L2
L2
L3
main

```

References

1. Geeks for Geeks. (2024). Three address code in Compiler. [Online]. geeksforgeeks. Last Updated: 27 September 2024. Available at:
<https://www.geeksforgeeks.org/three-address-code-compiler/> [Accessed 23 November 2024].
2. Geeks for Geeks. (2024). Symbol Table in Compiler. [Online]. geeksforgeeks. Last Updated: 27 September 2024. Available at:
<https://www.geeksforgeeks.org/symbol-table-compiler/> [Accessed 23 November 2024].