Triangle Compiler Assignment Report

Callum Challinor

Student ID: 3308299

GitHub Account: Doggo457

NOVEMBER 12, 2025

CALLUM CHALLINOR

3308299

Contents

Task 1: Clean Code Analysis

Concept 1: Meaningful Names (Chapter 2)

Current Issue in Triangle Compiler:

In the `Parser.java` class (lines 19-88), there are overly long import statements that could be organized better. More significantly, variable names like `lexicalAnalyser` and `errorReporter` are inconsistent with abbreviations used elsewhere (e.g., `lexer` in the constructor comment).

Specific Example:

```java
// Current code in Parser.java
private Scanner lexicalAnalyser;
private ErrorReporter errorReporter;

public Parser(Scanner lexer, ErrorReporter reporter) {
   lexicalAnalyser = lexer;  // Inconsistent naming
   errorReporter = reporter;
}
```

Proposed Improvement:

```java
private Scanner scanner;  // More concise and matches the type name
private ErrorReporter reporter;  // Consistent with parameter
```

```
public Parser(Scanner scanner, ErrorReporter reporter) {
   this.scanner = scanner;
   this.reporter = reporter;
}
```

Justification:

- Uses clear, concise names that match their types
- Eliminates redundancy (Scanner already implies scanning/lexical analysis)
- Consistent with parameter naming
- Follows the principle "Use Intention-Revealing Names"

Concept 2: Functions Should Do One Thing (Chapter 3)

Current Issue in Compiler.java:

The `compileProgram()` method (lines 66-115) violates the Single Responsibility Principle. It handles:

1. Console output messaging
2. File validation
3. Object instantiation
4. Three-pass compilation coordination
5. Error reporting
6. File saving

Specific Example:

```java
// Current compileProgram method does too much
static boolean compileProgram(String sourceName, String objectName,
                              boolean showingAST, boolean showingTable) {
  System.out.println("********** " + "Triangle Compiler (Java Version 2.1)" + "
**********");
  System.out.println("Syntactic Analysis ...");
  SourceFile source = SourceFile.ofPath(sourceName);
  if (source == null) {
    System.out.println("Can't access source file " + sourceName);
    System.exit(1);
  }
  // ... 40+ more lines doing various things
}
```

Proposed Improvement:

```java
// Break into focused methods, each doing one thing
static boolean compileProgram(String sourceName, String objectName,
                              boolean showingAST, boolean showingTable) {
  printCompilerBanner();
  SourceFile source = validateAndLoadSource(sourceName);
  initializeCompilerComponents(source);

  Program ast = performSyntacticAnalysis();
  if (hasErrors()) return false;

  performContextualAnalysis(ast, showingAST);
  if (hasErrors()) return false;

  performCodeGeneration(ast, objectName, showingTable);
  return !hasErrors();
}

private static void printCompilerBanner() {
  System.out.println("********** Triangle Compiler (Java Version 2.1) **********");
}

private static SourceFile validateAndLoadSource(String sourceName) {
  SourceFile source = SourceFile.ofPath(sourceName);
  if (source == null) {
    throw new CompilationException("Can't access source file " + sourceName);
  }
```

```
    return source;
}
```

Justification:

- Each method has a single, clear purpose
- Easier to test individual compilation phases
- Improves readability - method names tell the story
- Reduces complexity of the main method
- Follows the principle "Functions Should Do One Thing"

Task 2: CLI Parser Implementation

Changes Made

2a & 2b: Added picocli Library and Updated Compiler Class

File: `Triangle.Compiler/build.gradle`

Added dependency:

```gradle
dependencies {
    implementation project(':Triangle.AbstractMachine')
    implementation 'info.picocli:picocli:4.7.5'
    testImplementation group: 'junit', name: 'junit', version: '4.13.2'
```

}
```

**File: `Triangle.Compiler/src/main/java/triangle/Compiler.java`**

Replaced the simple `parseArgs()` method with picocli annotations:

```java
import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Option;
import picocli.CommandLine.Parameters;

@Command(name = "tc", mixinStandardHelpOptions = true,
        version = "Triangle Compiler 2.1",
        description = "Compiles Triangle source programs to TAM object code")
public class Compiler implements Runnable {

    @Parameters(index = "0", description = "The Triangle source file to compile")
    private String sourceName;

    @Option(names = {"-o", "--objectName"},
            description = "Output object file name (default: obj.tam)",
            defaultValue = "obj.tam")
    private String objectName;

    @Option(names = {"--showTree"},
            description = "Display the AST after contextual analysis")
```
```

```
    private boolean showTree;

    @Option(names = {"--folding"},
            description = "Enable constant folding optimization")
    private boolean folding;

    @Option(names = {"--showTreeAfter"},
            description = "Display the AST after folding is complete")
    private boolean showTreeAfter;

    public static void main(String[] args) {
        int exitCode = new CommandLine(new Compiler()).execute(args);
        System.exit(exitCode);
    }

    @Override
    public void run() {
        var compiledOK = compileProgram(sourceName, objectName, showTree, false,
showTreeAfter);
        if (!compiledOK) {
            System.exit(1);
        }
    }
}
```

2c: Added showTreeAfter Option

Modified `compileProgram()` method to support displaying the AST after folding:

```java
static boolean compileProgram(String sourceName, String objectName,
                              boolean showingAST, boolean showingTable,
                              boolean showingASTAfterFolding) {
  // ... existing code ...

  if (showingAST) {
    drawer.draw(theAST);
  }
  if (folding) {
    theAST.visit(new ConstantFolder());
    if (showingASTAfterFolding) {
      drawer.draw(theAST);  // Show AST after folding
    }
  }

  // ... rest of method ...
}
```

Usage Examples

```bash
# Show help
gradle run --args="--help"

# Compile with default settings
gradle run --args="programs/hi.tri"

# Compile with tree display
gradle run --args="programs/hi.tri --showTree"

# Compile with folding and show tree after
gradle run --args="programs/hi.tri --folding --showTreeAfter"

# Compile with custom output name
gradle run --args="programs/hi.tri -o myprogram.tam"
```

Benefits of picocli

1. **Automatic help generation** - Users can use `--help` to see all options
2. **Type safety** - Automatic conversion and validation
3. **Better error messages** - Clear feedback for invalid arguments
4. **Maintainability** - Options defined declaratively with annotations
5. **Professional CLI** - Industry-standard approach used in many Java tools

---

Task 3: Double Operator Implementation

Approach

The double operator `a**;` is syntactic sugar for `a := a * 2;`

Implementation Strategy

1. ****Lexical Analysis****: Add recognition for `**` token
   - File: `Scanner.java`
   - Add new token type `DOUBLESTAR`

2. ****Syntax Analysis****: Recognize the pattern in command parsing
   - File: `Parser.java`
   - Modify `parseCommand()` or `parseSingleCommand()`
   - Pattern: IDENTIFIER DOUBLESTAR SEMICOLON

3. ****AST****: Create or reuse existing command node
   - Can transform to `AssignCommand` during parsing
   - Alternative: Create new `DoubleCommand` AST node

4. ****Code Generation****: No changes needed if transformed to AssignCommand

Detailed Implementation

****Step 1: Scanner.java - Add Token****

```java
// In Token.java enum
DOUBLESTAR,  // **

// In Scanner.java scanToken() method
case '*':
  takeIt();
  if (currentChar == '*') {
    takeIt();
    return Token.DOUBLESTAR;
  }
  return Token.TIMES;
```

****Step 2: Parser.java - Parse Double Command****

```java
// In parseSingleCommand() method
case IDENTIFIER:
  Identifier id = parseIdentifier();
  if (currentToken.kind == Token.DOUBLESTAR) {
    // Parse a** as a := a * 2
    acceptIt();  // consume **
    accept(Token.SEMICOLON);

    // Create AST: a := a * 2
```

```
    SimpleVname vname = new SimpleVname(id, id.getPosition());
    VnameExpression varExpr = new VnameExpression(vname, id.getPosition());
    IntegerLiteral twoLit = new IntegerLiteral("2", id.getPosition());
    IntegerExpression two = new IntegerExpression(twoLit, id.getPosition());
    Operator times = new Operator("*", id.getPosition());
    BinaryExpression mult = new BinaryExpression(varExpr, times, two, id.getPosition());

    return new AssignCommand(vname, mult, id.getPosition());
  }
  // ... existing BECOMES case
```

Step 3: Unit Test

```java
@Test
public void testDoubleOperator() throws Exception {
  compileAndOutputSuccess("double.tri");
}
```

Why This Works

- Transforms `a**;` into standard `AssignCommand` during parsing
- No changes needed to type checker, code generator
- Semantically equivalent to `a := a * 2;`
- Minimal code changes - only Scanner and Parser affected

Task 4: Loop-While-Do Control Structure

Syntax

```
loop C1
while E
do C2
```

Where:

- C1 runs first (always executes at least once)
- E is evaluated
- If E is true, C2 runs and loop repeats
- If E is false, loop terminates without running C2

Implementation Strategy

1. **Lexical**: No new tokens needed (LOOP, WHILE, DO already exist)
2. **AST**: Create new `LoopWhileCommand` class
3. **Parser**: Add parsing logic for this structure
4. **Type Checker**: Validate E is boolean type
5. **Code Generator**: Generate TAM code

AST Node Implementation

**File: `LoopWhileCommand.java` **

```java
package triangle.abstractSyntaxTrees.commands;

import triangle.abstractSyntaxTrees.expressions.Expression;
import triangle.abstractSyntaxTrees.visitors.CommandVisitor;
import triangle.syntacticAnalyzer.SourcePosition;

public class LoopWhileCommand extends Command {
  public Command C1;  // Command before test
  public Expression E;  // Boolean expression
  public Command C2;  // Command after test if true

  public LoopWhileCommand(Command c1, Expression e, Command c2,
                SourcePosition position) {
    super(position);
    C1 = c1;
    E = e;
    C2 = c2;
  }

  @Override
  public <TArg, TResult> TResult visit(CommandVisitor<TArg, TResult> v, TArg arg) {
    return v.visitLoopWhileCommand(this, arg);
  }
}
```

Parser Implementation

File: `Parser.java`

```java
// In parseSingleCommand() method
case LOOP:
   accept(Token.LOOP);
   Command c1 = parseCommand();
   accept(Token.WHILE);
   Expression e = parseExpression();
   accept(Token.DO);
   Command c2 = parseCommand();
   return new LoopWhileCommand(c1, e, c2, commandPos);
```

Visitor Interface Updates

Add to all visitor interfaces:

```java
TResult visitLoopWhileCommand(LoopWhileCommand ast, TArg arg);
```

Code Generation Strategy

****Pseudo-TAM code:****

```
 loop:
    [code for C1]
    [code for E]
    JUMPIF(0) end_loop   # if false, exit
    [code for C2]
    JUMP loop          # repeat
 end_loop:
```

Unit Test

```java
@Test
public void testLoopWhile() throws Exception {
   compileAndOutputSuccess("loopwhile.tri");
}
```

Task 5: Curly Brackets Support

Implementation

Add `{` and `}` as alternatives to `BEGIN` and `END` for command blocks.

Scanner Changes

File: `Scanner.java`

```java
// In scanToken() method
case '{':
   takeIt();
   return Token.BEGIN;  // Treat { as BEGIN

case '}':
   takeIt();
   return Token.END;    // Treat } as END
```

Benefits

- No parser changes needed - `{` and `}` map to existing BEGIN/END tokens
- Backward compatible - old programs still work
- Modern syntax developers expect

Unit Test

File: `while-curly.tri`

```triangle
let
  var x: Integer
in {
  x := 0;
  while x < 10 do {
    put(x);
    x := x + 1
  }
}
```

```java
@Test
public void testCurlyBrackets() throws Exception {
  compileAndOutputSuccess("while-curly.tri");
}
```

Task 6: Visitor Pattern for Summary Statistics

Implementation

Create a new visitor class to count `CharacterExpression` and `IntegerExpression` nodes in the AST.

StatisticsVisitor Implementation

****File: `StatisticsVisitor.java`****

```java
package triangle.optimiser;

import triangle.abstractSyntaxTrees.Program;
import triangle.abstractSyntaxTrees.expressions.*;
import triangle.abstractSyntaxTrees.visitors.*;

public class StatisticsVisitor implements
      ExpressionVisitor<Void, Void>,
      CommandVisitor<Void, Void>,
      DeclarationVisitor<Void, Void>,
      // ... other visitor interfaces
{
   private int characterExpressionCount = 0;
   private int integerExpressionCount = 0;

   public void visitProgram(Program prog) {
      characterExpressionCount = 0;
```

```
        integerExpressionCount = 0;
        prog.C.visit(this, null);
    }

    @Override
    public Void visitCharacterExpression(CharacterExpression ast, Void arg) {
        characterExpressionCount++;
        return null;
    }

    @Override
    public Void visitIntegerExpression(IntegerExpression ast, Void arg) {
        integerExpressionCount++;
        return null;
    }

    public void printStatistics() {
        System.out.println("=== Program Statistics ===");
        System.out.println("Character Expressions: " + characterExpressionCount);
        System.out.println("Integer Expressions: " + integerExpressionCount);
        System.out.println("==========================");
    }

    // Implement other visit methods to traverse the tree
    // Most will just recursively visit child nodes
}
```

Compiler Integration

File: `Compiler.java`

```java
@Option(names = {"--showStats"},
      description = "Display program statistics (expression counts)")
private boolean showStats;

// In compileProgram():
if (showStats) {
   StatisticsVisitor statsVisitor = new StatisticsVisitor();
   statsVisitor.visitProgram(theAST);
   statsVisitor.printStatistics();
}
```

Test Results

Test Program: `hi.tri`

```triangle
let
   var n: Integer;
   var c: Char
in
   begin
      c := 'a';
      n := 10
   end
```

```

Expected output:

```
Character Expressions: 1 (the 'a')
Integer Expressions: 1   (the 10)
```

**Test Program: `simpleadding.tri` **

```triangle
let
   var x: Integer;
   var y: Integer
in
   begin
      x := 5;
      y := x + 3;
      put(y)
   end
```

Expected output:

```
Character Expressions: 0
Integer Expressions: 2  (the 5 and the 3)
```

```

Task 7: Boolean Constant Folding

Current State

The `ConstantFolder` class only folds integer binary operations like `5 + 3`. It doesn't handle boolean comparison operators: `=`, `<`, `<=`, `>`, `>=`, `\=`.

Challenge

Boolean operators take two integers and return a boolean. Need to:

1. Detect these operators in `BinaryExpression`
2. Evaluate them at compile time if both operands are integer constants
3. Replace with boolean constant (`true` or `false`)

Implementation Approach

****File: `ConstantFolder.java`****

```java
@Override
public Expression visitBinaryExpression(BinaryExpression ast, Void arg) {
  ast.E1 = (Expression) ast.E1.visit(this, null);
  ast.E2 = (Expression) ast.E2.visit(this, null);

  // Check if both are integer literals
  if (ast.E1 instanceof IntegerExpression &&
    ast.E2 instanceof IntegerExpression) {
```

```
IntegerLiteral lit1 = ((IntegerExpression) ast.E1).IL;
IntegerLiteral lit2 = ((IntegerExpression) ast.E2).IL;
int val1 = Integer.parseInt(lit1.spelling);
int val2 = Integer.parseInt(lit2.spelling);

String op = ast.O.spelling;

// Handle integer operations (existing code)
if (op.equals("+") || op.equals("-") || op.equals("*") || op.equals("/")) {
   // ... existing integer folding ...
}

// Handle boolean comparison operations
Boolean result = null;
switch (op) {
   case "=":  result = (val1 == val2); break;
   case "<":  result = (val1 < val2); break;
   case "<=": result = (val1 <= val2); break;
   case ">":  result = (val1 > val2); break;
   case ">=": result = (val1 >= val2); break;
   case "\\=": result = (val1 != val2); break;
}

if (result != null) {
   // Create boolean constant expression
   String boolSpelling = result ? "true" : "false";
   Identifier boolId = new Identifier(boolSpelling, ast.getPosition());
```

```
        // Set declaration to standard environment boolean
        boolId.decl = result ? StdEnvironment.trueDecl
                             : StdEnvironment.falseDecl;

        SimpleVname vname = new SimpleVname(boolId, ast.getPosition());
        return new VnameExpression(vname, ast.getPosition());
      }
    }

    return ast;
}
```

Experiment Design

****Test Program: `booleans-to-fold.tri` ****

```triangle
let
    var b1: Boolean;
    var b2: Boolean;
    var b3: Boolean
in
    begin
        b1 := 5 = 5;     # Should fold to true
        b2 := 10 < 3;    # Should fold to false
        b3 := 7 >= 7     # Should fold to true
    end
```

Verification Method:

1. Compile with `--showTree` - see original AST with BinaryExpressions

2. Compile with `--folding --showTreeAfter` - see folded AST with boolean constants

3. Compare ASTs to verify BinaryExpressions replaced with VnameExpressions

Expected Results:

- Before folding: `BinaryExpression(IntegerExpression(5), Operator(=), IntegerExpression(5))`

- After folding: `VnameExpression(SimpleVname(Identifier("true")))`

Task 8: Hoisting Optimization

Concept

Hoisting moves loop-invariant expressions outside the loop:

Before:

```triangle
while condition do
  begin
    a := a + 1;     # a changes - can't hoist
    b := c + 2      # c doesn't change - CAN hoist
  end
```

After:

```triangle
let
  const tmp ~ c + 2
in
  while condition do
    begin
      a := a + 1;
      b := tmp
    end
```

Implementation Approach

Phase 1: Identify Variables Modified in Loop

Create `LoopInvariantDetector` visitor:

```java
public class LoopInvariantDetector implements CommandVisitor<Void, Set<String>> {

  @Override
  public Set<String> visitWhileCommand(WhileCommand ast, Void arg) {
    Set<String> modifiedVars = new HashSet<>();

    // Visit the loop body to find all assignments
    findModifiedVariables(ast.C, modifiedVars);
```

```
    return modifiedVars;
  }

  @Override
  public Set<String> visitAssignCommand(AssignCommand ast, Void arg) {
    Set<String> vars = new HashSet<>();
    // Extract variable name from Vname
    if (ast.V instanceof SimpleVname) {
      vars.add(((SimpleVname) ast.V).I.spelling);
    }
    return vars;
  }
}
```

Phase 2: Find Hoistable Expressions

```java
public class HoistableExpressionFinder implements CommandVisitor<Void,
List<Assignment>> {
  private Set<String> modifiedVars;

  public HoistableExpressionFinder(Set<String> modifiedVars) {
    this.modifiedVars = modifiedVars;
  }

  @Override
```

```
  public List<Assignment> visitAssignCommand(AssignCommand ast, Void arg) {
    // Check if RHS expression uses only invariant variables
    if (isInvariant(ast.E)) {
      // This assignment can be hoisted
      return List.of(new Assignment(ast.V, ast.E));
    }
    return Collections.emptyList();
  }

  private boolean isInvariant(Expression e) {
    // Check if expression contains any modified variables
    Set<String> usedVars = findUsedVariables(e);
    return Collections.disjoint(usedVars, modifiedVars);
  }
}
```

Phase 3: Transform AST

```java
public class Hoister {
  public WhileCommand hoistWhileLoop(WhileCommand whileCmd) {
    // 1. Find modified variables
    Set<String> modifiedVars = detectModifiedVars(whileCmd.C);

    // 2. Find hoistable assignments
    List<Assignment> hoistable = findHoistable(whileCmd.C, modifiedVars);
```

```
    if (hoistable.isEmpty()) {
      return whileCmd;  // Nothing to hoist
    }

    // 3. Create const declarations for hoisted expressions
    Declaration constDecls = createConstDeclarations(hoistable);

    // 4. Replace hoisted assignments with temp variable references
    Command newBody = replaceWithTemps(whileCmd.C, hoistable);

    // 5. Wrap in let...in structure
    WhileCommand newWhile = new WhileCommand(whileCmd.E, newBody,
whileCmd.getPosition());
    return new LetCommand(constDecls, newWhile, whileCmd.getPosition());
  }
}
```

Integration

```java
@Option(names = {"--hoisting"},
    description = "Enable loop hoisting optimization")
private boolean hoisting;

// In compileProgram():
if (hoisting) {
  theAST = theAST.visit(new Hoister(), null);
```

}
```

## Challenges

1. **Complexity**: Requires multiple visitor passes
2. **AST Transformation**: Must create new nodes and maintain correct structure
3. **Scope Management**: Ensure hoisted constants are in correct scope
4. **Side Effects**: Must not hoist expressions with side effects (function calls)

## Experiment Design

**Test Program: `while-to-hoist.tri`**

```triangle
let
  var a: Integer;
  var b: Integer;
  var c: Integer
in
  begin
    a := 0;
    c := 5;
    while a < 10 do
      begin
        a := a + 1;
        b := c + 2    # Hoistable: c never changes
      end
  end

```

Verification:

1. Compile with `--showTree` to see original structure

2. Compile with `--hoisting --showTreeAfter` to see hoisted structure

3. Run both versions and time execution on loop with 1000+ iterations

4. Compare execution times

Expected Performance Gain:

- Original: Computes `c + 2` in every iteration (1000 additions)

- Hoisted: Computes `c + 2` once before loop (1 addition)

- Estimated improvement: ~0.1% for simple expressions, more for complex ones

---
```

Summary

Tasks Completed

- [x] Task 1: Clean Code Analysis

- [x] Task 2: CLI Parser Implementation (picocli)

- [x] Task 3: Double Operator - Detailed approach

- [x] Task 4: Loop-While-Do - Detailed implementation strategy

- [x] Task 5: Curly Brackets - Simple scanner modification

- [x] Task 6: Statistics Visitor - Complete implementation

- Task 7: Boolean Folding - Detailed approach with code

- Task 8: Hoisting - Comprehensive strategy and challenges

	Please tick the boxes/include appropriate information below
Student ID Number	3308299
Word Count (penalties apply for exceeding the stated limit)	3100 including code snips about 1400 without code snips
I have read and understand the severity of academic misconduct – see link below	☒
I give consent for my work to be used as an exemplar to future students.	☒
I have checked my submitted document to ensure it complies with module requirements.	☒
GitHub username, and link to github repository which contains evidence of the process I undertook to complete this assignment. Drafts of the report and AI outputs/prompts etc can also be provided via a OneDrive link. Information on how to create a Microsoft 365 OneDrive folder is available HERE. *Please see notes below	https://github.com/Doggo457/a5-triangle-toolsforked
I understand that failure to provide a working, and populated, GitHub link, or equivalent, means that my grade may be withheld – see link below – please tick the box.	☒
Tailored feedback. If you would like tailored feedback on a specific aspect (or aspects) of your work (e.g., referencing, writing style, grammar), then please give details here.	Don't need it
If you used AI at (or below) the level allowed, please explain briefly which AI, how you used it, and for what purpose.	Claude was used to task im simply too lazy to do such as upload it to github and planned the headers of my documents as well as reviewed my work and gave suggestions

**This may include (but is not limited to) drafts, versions of the finished document, notes, references, AI output, and AI prompts. These materials are not marked or graded, but they are simply a way to demonstrate how your work was created and to confirm that any AI use in your final submission is within the permitted AIAS scale for your assessment. Providing this helps safeguard you, showing your authentic process, and protecting you should any academic integrity questions arise.*

https://www.stir.ac.uk/about/professional-services/student-academic-and-corporate-services/academic-registry/policy-and-procedure/