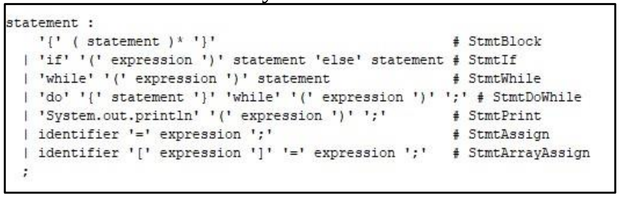
IMPLEMENTING DO/WHILE LOOP FUNCTIONALITY IN BABYCINO MINIJAVA COMPILER

1. INTRODUCTION

The following study documents the implementation of *do/while* loops, in the simple MiniJava compiler: babycino. Whilst *while* loops are supported by the compiler, *do/while* are not, however due to the similarity between these loops, *do/while* implementation should be simple. The project was completed using Windows operating system, Eclipse IDE to modify babycino classes, and Antlr4 to generate compiler files.

2. PARSING AND SEMANTIC ANALYSIS

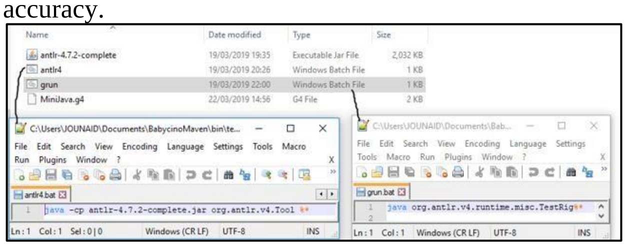
Initially, I must edit the *MiniJava.g4* grammar file to include a line for *do/while.* This is required to generate new parser and lexer classes from Antlr4 that include the *do/while* functionality.



```
statement :
    '{' ( statement )* '}'                                # StmtBlock
  | 'if' '(' expression ')' statement 'else' statement   # StmtIf
  | 'while' '(' expression ')' statement                  # StmtWhile
  | 'do' '{' statement '}' 'while' '(' expression ')' ';' # StmtDoWhile
  | 'System.out.println' '(' expression ')' ';'          # StmtPrint
  | identifier '=' expression ';'                         # StmtAssign
  | identifier '[' expression ']' '=' expression ';'      # StmtArrayAssign
  ;
```

Figure 1. Statement grammar block

As you can see in figure 1, I decided to copy the provided grammar for *while* statements, but with *statement* put before *'while',* and *'do'* added at the start. Even though not necessary, I enclosed *statement* curly brackets and have a semicolon at the end to ensure its accuracy.



```
Name                  Date modified      Type                 Size
antlr-4.7.2-complete  19/03/2019 19:35   Executable Jar File  2,032 KB
antlr4                19/03/2019 20:26   Windows Batch File   1 KB
grun                  19/03/2019 22:00   Windows Batch File   1 KB
MiniJava.g4           22/03/2019 14:56   G4 File              2 KB
```

C:\Users\JOUNAID\Documents\BabycinoMaven\bin\te...

antlr4.bat

```
java -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool %*
```

C:\Users\JOUNAID\Documents\Bab...

grun.bat

```
java org.antlr.v4.runtime.misc.TestRig %*
```

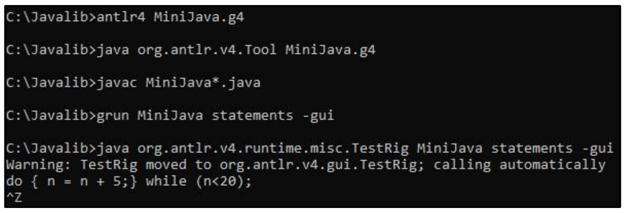
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Ln : 1 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS
Ln : 1 Col : 1 Windows (CR LF) UTF-8 INS

Figure 2. Bat files, antlr program, MiniJava.g4

In order to run antlr to generate the required files on windows, I have created two bat files:

antlr4.bat: java -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool %*

grun.bat: java -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool %* *grun.bat* will help when generating a parser tree whilst *antlr4.bat* is used when generating files.



```
C:\Javalib>antlr4 MiniJava.g4

C:\Javalib>java org.antlr.v4.Tool MiniJava.g4

C:\Javalib>javac MiniJava*.java

C:\Javalib>grun MiniJava statements -gui

C:\Javalib>java org.antlr.v4.runtime.misc.TestRig MiniJava statements -gui
Warning: TestRig moved to org.antlr.v4.gui.TestRig; calling automatically
do { n = n + 5;} while (n<20);
^Z
```

Figure 3. Parser tree/file generation commands

I changed the directory to *C:\Javalib* to make it easier to set the *CLASSPATH* (through windows environment variables) and copied across the grammar and bat files. The first three lines of the CMD shown in figure 3 are used to generate all the required files. The final command is used to generate an image representation of a parse tree for the following statement: do { n = n + 5;} while (n<20);

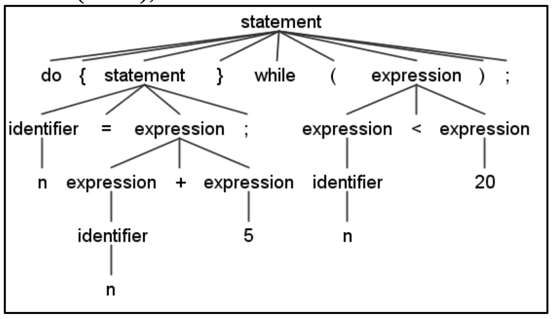
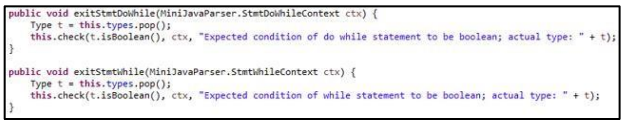


Figure 4. Parse tree

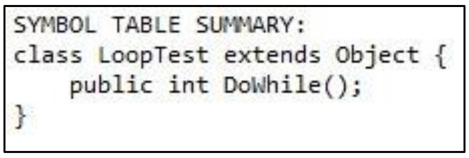
Figure 4 gives us a visual representation of the statement structure, and how the *do/while* loops are parsed, by splitting the statement into its constituent pieces. As this has been generated successfully I can conclude that the grammar statement for *do/while* loop implementation to be correct.



```
public void exitStmtDoWhile(MiniJavaParser.StmtDoWhileContext ctx) {
    Type t = this.types.pop();
    this.check(t.isBoolean(), ctx, "Expected condition of do while statement to be boolean; actual type: " + t);
}

public void exitStmtWhile(MiniJavaParser.StmtWhileContext ctx) {
    Type t = this.types.pop();
    this.check(t.isBoolean(), ctx, "Expected condition of while statement to be boolean; actual type: " + t);
}
```

Figure 5. Type checker (semantic analysis)

Figure 5 shows the modifications done to the *TypeChecker.java* class. Since both *while* and *do/while* loops are almost identical in functionality (with only their structures differing), I simply needed to copy the previous while type checker method (adding 'do' in front of each instance of while). The method essentially checks if the condition statement is Boolean, and if not will notify the user of the incorrect type used, before exiting the loop.



```
SYMBOL TABLE SUMMARY:
class LoopTest extends Object {
    public int DoWhile();
}
```

Figure 6. Symbol table summary

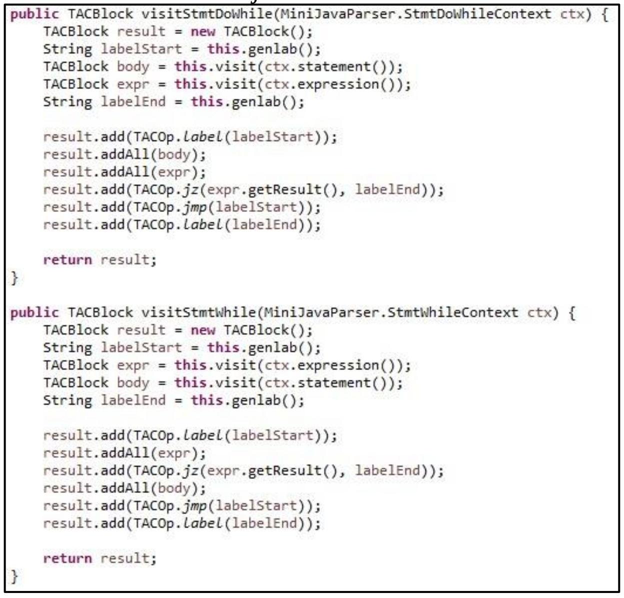
As further proof, figure 6 presents a screenshot from the console output (of a test run) showing the symbol table summary. This basically means that the program understands the syntax correctly from the parser.

3. CODE GENERATION

Code generation (in this context) is where the mini java code is converted into C code, which is analogous to human language translation.

Babycino uses TAC (three-address code) for intermediate code generation. Each instruction must have at most three operands, which generally include a combination of an assignment and a binary operator.

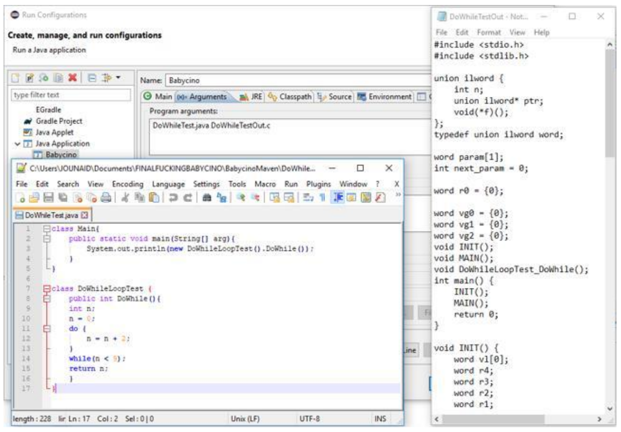
In order to generate correct code for *do/while* loops, I must edit the *TACGenerator.java* class.



```
public TACBlock visitStmtDoWhile(MiniJavaParser.StmtDoWhileContext ctx) {
    TACBlock result = new TACBlock();
    String labelStart = this.genlab();
    TACBlock body = this.visit(ctx.statement());
    TACBlock expr = this.visit(ctx.expression());
    String labelEnd = this.genlab();

    result.add(TACOp.label(labelStart));
    result.addAll(body);
    result.addAll(expr);
    result.add(TACOp.jz(expr.getResult(), labelEnd));
    result.add(TACOp.jmp(labelStart));
    result.add(TACOp.label(labelEnd));

    return result;
}

public TACBlock visitStmtWhile(MiniJavaParser.StmtWhileContext ctx) {
    TACBlock result = new TACBlock();
    String labelStart = this.genlab();
    TACBlock expr = this.visit(ctx.expression());
    TACBlock body = this.visit(ctx.statement());
    String labelEnd = this.genlab();

    result.add(TACOp.label(labelStart));
    result.addAll(expr);
    result.add(TACOp.jz(expr.getResult(), labelEnd));
    result.addAll(body);
    result.add(TACOp.jmp(labelStart));
    result.add(TACOp.label(labelEnd));

    return result;
}
```

Figure 7. TAC generator

Figure 7 shows the new TAC generator method for *do/while* implementation in the above block, and the preexisting method for *while* loops in the below block. As mentioned previously, these two types of loops are incredibly similar, so I initially just copied the *while* block (adding 'do' before each instance of 'while'). However, since *do/while* loops run the code before condition statement, I have placed the *body* add statement in front of the expression add statement. Since the expression checker line (*...(expr.getResult(), labelEnd)…)* is now after the *body*, I know that the *body* code will be generated at least one time before checking the condition statement, meaning this method should implement *do/while* loop functionality correctly.



Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
EGradle
Gradle Project
Java Applet
Java Application
Babycino
Main
Arguments
JRE
Classpath
Source
Environment
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
DoWhileTest.java
length : 228 lir Ln : 17 Col : 2 Sel : 0 | 0 Unix (LF) UTF-8 INS
DoWhileTestOut - Not...
File Edit Format View Help

Name: Babycino

Program arguments: DoWhileTest.java DoWhileTestOut.c

C:\Users\JOUNAID\Documents\FINALFUCKINGBABYCINO\BabycinoMaven\DoWhile...

```
class Main{
    public static void main(String[] arg){
        System.out.println(new DoWhileLoopTest().DoWhile());
    }
}

class DoWhileLoopTest {
    public int DoWhile(){
    int n;
    n = 0;
    do {
        n = n + 2;
    }
    while(n < 9);
    return n;
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>

union ilword {
    int n;
    union ilword* ptr;
    void(*f)();
};
typedef union ilword word;

word param[1];
int next_param = 0;

word r0 = {0};

word vg0 = {0};
word vg1 = {0};
word vg2 = {0};
void INIT();
void MAIN();
void DoWhileLoopTest_DoWhile();
int main() {
    INIT();
    MAIN();
    return 0;
}

void INIT() {
    word vl[0];
    word r4;
    word r3;
    word r2;
    word r1;
```

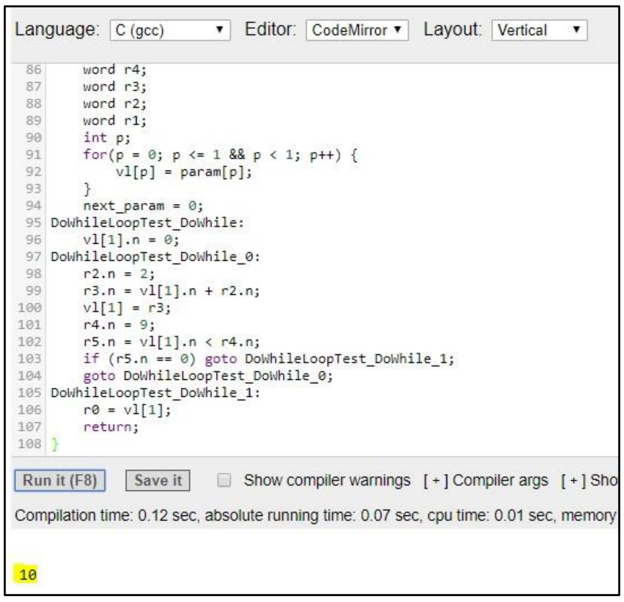
Figure 8. Testing compiler

To test whether *do/while* loops are compiled properly, I created a simple program (following syntax from appel):

```
class Main{
public static void main(String[] arg){
System.out.println(new DoWhileLoopTest().DoWhile());
                }
}  class DoWhileLoopTest {
public int DoWhile(){
int n;                 n = 0;
do {
                                n = n + 2;
                }
while(n < 9);
return n;
                }
}
```

The above program declares a variable, *n*, and will keep adding 2 to that variable whilst *n* is smaller than 9. As soon as *n* is greater than 9, it will return *n*. The expected result is 10.

Using eclipse IDE to set the arguments (to the corresponding in/out file names), C code was successfully generated (as shown in figure 8).



Language: C (gcc) Editor: CodeMirror Layout: Vertical

```
    word r4;
    word r3;
    word r2;
    word r1;
    int p;
    for(p = 0; p <= 1 && p < 1; p++) {
        vl[p] = param[p];
    }
    next_param = 0;
DoWhileLoopTest_DoWhile:
    vl[1].n = 0;
DoWhileLoopTest_DoWhile_0:
    r2.n = 2;
    r3.n = vl[1].n + r2.n;
    vl[1] = r3;
    r4.n = 9;
    r5.n = vl[1].n < r4.n;
    if (r5.n == 0) goto DoWhileLoopTest_DoWhile_1;
    goto DoWhileLoopTest_DoWhile_0;
DoWhileLoopTest_DoWhile_1:
    r0 = vl[1];
    return;
}
```

Compilation time: 0.12 sec, absolute running time: 0.07 sec, cpu time: 0.01 sec, memory

```
10
```

Figure 9. Compiled C code test

Using an online gcc compiler (shown in figure 9), I can conclude that the generated C codesuccessfully compiles, and gives the correct output.

Intermediate TAC code generated:

```
INIT:     r1 = 1     r2 = 0     vg0 =
malloc r2     r3 = vg0     r2 = 0     vg1
= malloc r2     r3 = vg1     r2 = 1
vg2 = malloc r2     r3 = vg2     r4 =
DoWhileLoopTest.DoWhile     [r3] = r4
r3 = r3 offset r1     return MAIN:     r1
= 1     r2 = malloc r1     [r2] = vg2
r3 = [r2]     r4 = 0     r5 = r3 offset
r4     r6 = [r5]     param r2     call r6
r7 = r0     write r7     return
DoWhileLoopTest.DoWhile:
    r1 = 0     vl1 = r1 DoWhileLoopTest.DoWhile@0:
r2 = 2     r3 = vl1 + r2     vl1 = r3     r4 = 9
r5 = vl1 < r4     if (r5=0) jmp
DoWhileLoopTest.DoWhile@1     jmp
DoWhileLoopTest.DoWhile@0 DoWhileLoopTest.DoWhile@1:
    r0 = vl1     return
```

Figure 10. Peephole optimisation

Figure 10 shows a small addition to the peephole optimiser, which essentially checks whether r1 = 0, and if so, remove the nth + 1 instruction.

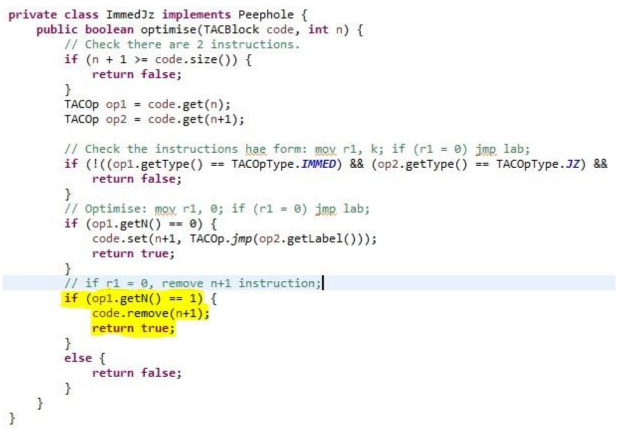
Comparing the generated code from before to after optimisation, all dead and inefficient instructions have been removed resulting in a significant reduction in size. After analysing the optimised code I can conclude that no further optimisation is required, as the compiler is fully optimised in its current form.

5. TESTING

The first test that must be conducted is to see if the *build.bat* file correctly compiles the program. Please note when running this file, 'JAVA_HOME' path **MUST** be set correctly to JDK bin for it to compile.

4. OPTIMISATION

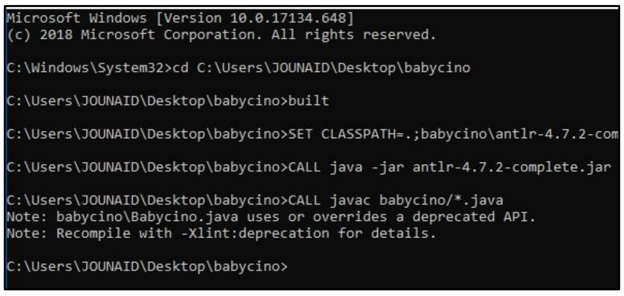
There are two classes already provided that will actively optimise the intermediate code for *do/while* loops, *TACPeepholeOptimiser* and *TACDeadCodeOptimiser*. In general, the peephole optimiser looks at small segments of code, called a ‹peephole› or ‹window›, and sees whether faster/shorter code can replace the segment of instructions. The dead code optimiser removes any redundant code that does not affect the overall processing or end result. This can drastically reduce the memory footprint.



```
private class ImmedJz implements Peephole {
    public boolean optimise(TACBlock code, int n) {
        // Check there are 2 instructions.
        if (n + 1 >= code.size()) {
            return false;
        }
        TACOp op1 = code.get(n);
        TACOp op2 = code.get(n+1);

        // Check the instructions hae form: mov r1, k; if (r1 = 0) jmp lab;
        if (!((op1.getType() == TACOpType.IMMED) && (op2.getType() == TACOpType.JZ) &&
            return false;
        }
        // Optimise: mov r1, 0; if (r1 = 0) jmp lab;
        if (op1.getN() == 0) {
            code.set(n+1, TACOp.jmp(op2.getLabel()));
            return true;
        }
        // if r1 = 0, remove n+1 instruction;
        if (op1.getN() == 1) {
            code.remove(n+1);
            return true;
        }
        else {
            return false;
        }
    }
}
```

Optimised Intermediate TAC code generated:

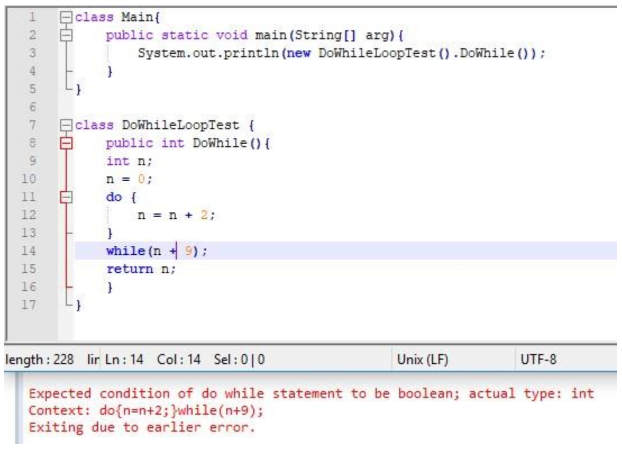
```
INIT:     r2 = 0
vg0 = malloc r2
r2 = 0     vg1 =
malloc r2     r2 =
1     vg2 = malloc
r2     r3 = vg2
    r4 =
DoWhileLoopTest.DoWhile
[r3] = r4     return MAIN:
r1 = 1     r2 = malloc r1
[r2] = vg2     r3 = [r2]     r4
= 0     r5 = r3 offset r4
r6 = [r5]     param r2     call
r6     r7 = r0     write r7
return DoWhileLoopTest.DoWhile:
    vl1 = 0 DoWhileLoopTest.DoWhile@0:
r2 = 2     r3 = vl1 + r2     vl1 = r3
r4 = 9     r5 = vl1 < r4     if (r5=0) jmp
DoWhileLoopTest.DoWhile@1     jmp
DoWhileLoopTest.DoWhile@0
DoWhileLoopTest.DoWhile@1:
    r0 = vl1
return
```



```
Microsoft Windows [Version 10.0.17134.648]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\System32>cd C:\Users\JOUNAID\Desktop\babycino

C:\Users\JOUNAID\Desktop\babycino>built

C:\Users\JOUNAID\Desktop\babycino>SET CLASSPATH=.;babycino\antlr-4.7.2-com

C:\Users\JOUNAID\Desktop\babycino>CALL java -jar antlr-4.7.2-complete.jar

C:\Users\JOUNAID\Desktop\babycino>CALL javac babycino/*.java
Note: babycino\Babycino.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\JOUNAID\Desktop\babycino>
```

Figure 11. Build.bat test

As you can see in figure 11, the build batch file correctly sets the 'CLASSPATH' before running *antlr4* and *javac* to generate and compile the files. This test is successful.



```java
class Main{
    public static void main(String[] arg){
        System.out.println(new DoWhileLoopTest().DoWhile());
    }
}

class DoWhileLoopTest {
    public int DoWhile(){
    int n;
    n = 0;
    do {
        n = n + 2;
    }
    while(n + 9);
    return n;
    }
}
```

```
Expected condition of do while statement to be boolean; actual type: int
Context: do{n=n+2;}while(n+9);
Exiting due to earlier error.
```

Figure 12. Type checker test

To unit test the type checker class, I edited the previous test java *do/while* file to have the condition as *'(n + 9)'*. As you can see in the console of figure 12, the correct error message is output. This test proves the type checkers functionality.

6. CONCLUSION

To conclude, I have successfully implemented *do/while* functionality into babycino mini java compiler. I have optimised the code to ensure maximum efficiency and minimum memory footprint when compiling. Finally, I created a build.bat file that will compile the babycino files.