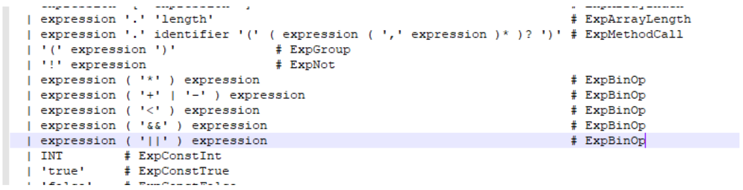
Module Code: CS2CO16
Assignment Report Title: Adding Boolean OR to a compiler
Student Number: 27002688
Date of completion:
Actual time spent on assignment:
Assignment evaluation (3 key points):

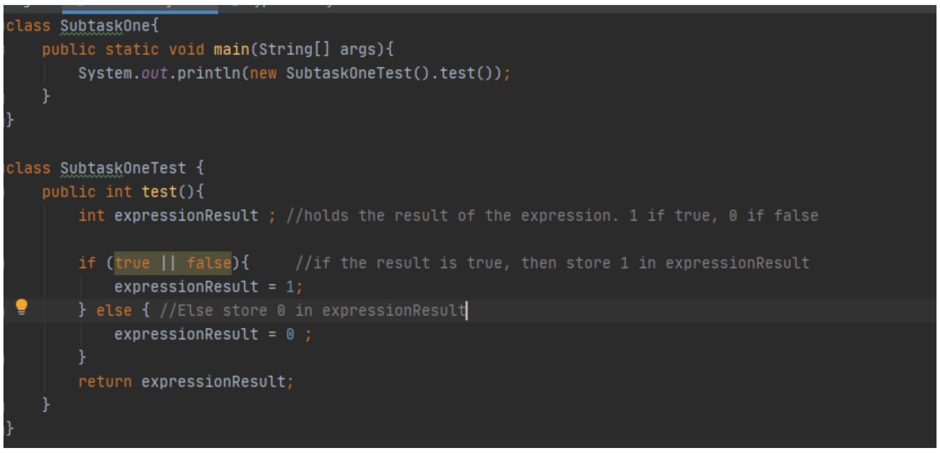
Subtask 1: Parsing (20%)

1. Screenshot

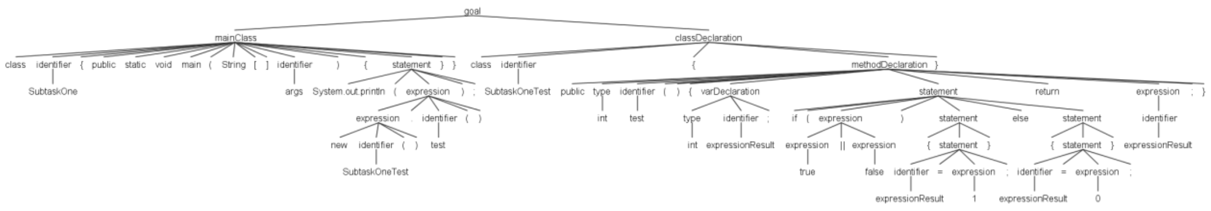


```
  | expression '.' 'length'                                     # ExpArrayLength
  | expression '.' identifier '(' ( expression ( ',' expression )* )? ')' # ExpMethodCall
  | '(' expression ')'                   # ExpGroup
  | '!' expression                       # ExpNot
  | expression ( '*' ) expression                                # ExpBinOp
  | expression ( '+' | '-' ) expression                          # ExpBinOp
  | expression ( '<' ) expression                                # ExpBinOp
  | expression ( '&&' ) expression                               # ExpBinOp
  | expression ( '||' ) expression                               # ExpBinOp
  | INT        # ExpConstInt
  | 'true'     # ExpConstTrue
```

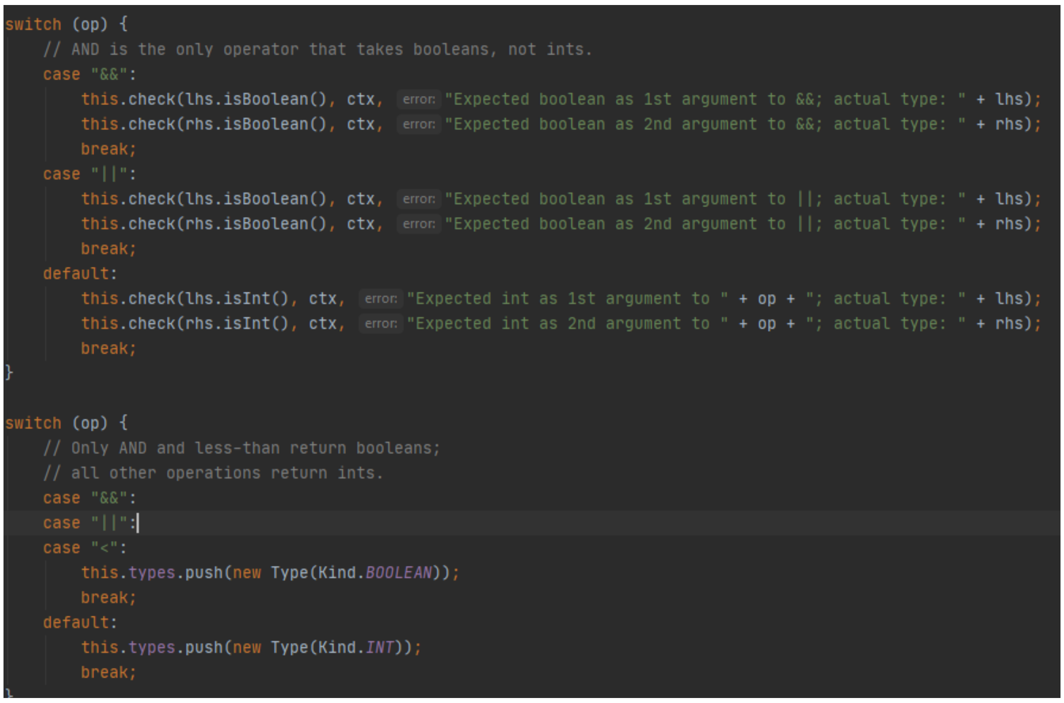
2. In order to modify babycino to parse || expressions, I had to modify the file MiniJava.g4 in order to accept || expressions.
3. Below is the code that contains a single || expression along with the parse tree for this program.



```
class SubtaskOne{
    public static void main(String[] args){
        System.out.println(new SubtaskOneTest().test());
    }
}

class SubtaskOneTest {
    public int test(){
        int expressionResult ; //holds the result of the expression. 1 if true, 0 if false

        if (true || false){     //if the result is true, then store 1 in expressionResult
            expressionResult = 1;
        } else { //Else store 0 in expressionResult
            expressionResult = 0 ;
        }
        return expressionResult;
    }
}
```


goal
mainClass
classDeclaration
class identifier { public static void main (String [ ] identifier) { statement } }
class identifier { methodDeclaration }
SubtaskOne
args System.out.println (expression) ;
SubtaskOneTest public type identifier ( ) { varDeclaration statement return expression ; }
expression . identifier ( )
int test type identifier ;
if (expression) statement else statement
identifier
new identifier ( ) test
int expressionResult
expression || expression
{ statement }
{ statement }
expressionResult
SubtaskOneTest
true false identifier = expression ; identifier = expression ;
expressionResult 1 expressionResult 0

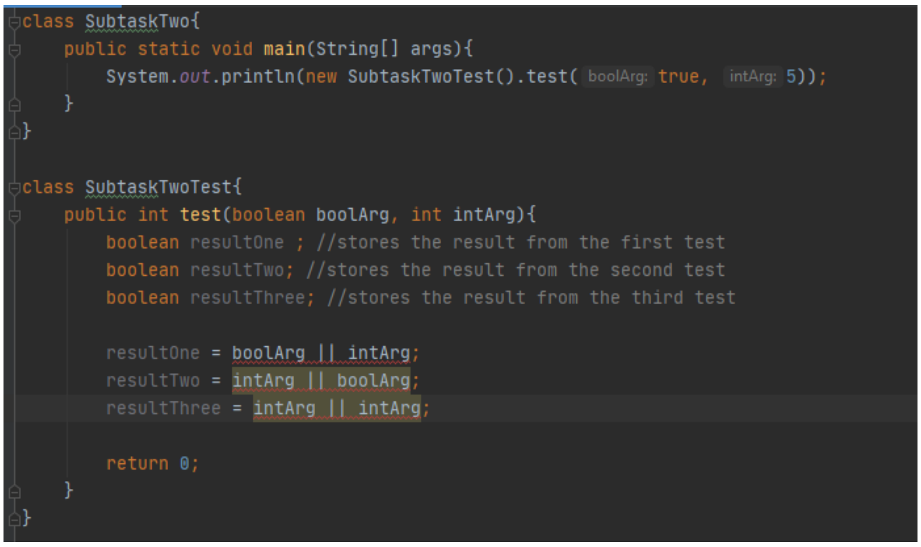
Subtask 2: Semantic Analysis

1. 

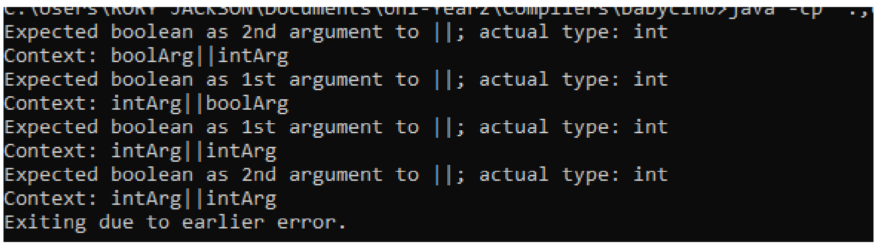
```
switch (op) {
    // AND is the only operator that takes booleans, not ints.
    case "&&":
        this.check(lhs.isBoolean(), ctx,  error: "Expected boolean as 1st argument to &&; actual type: " + lhs);
        this.check(rhs.isBoolean(), ctx,  error: "Expected boolean as 2nd argument to &&; actual type: " + rhs);
        break;
    case "||":
        this.check(lhs.isBoolean(), ctx,  error: "Expected boolean as 1st argument to ||; actual type: " + lhs);
        this.check(rhs.isBoolean(), ctx,  error: "Expected boolean as 2nd argument to ||; actual type: " + rhs);
        break;
    default:
        this.check(lhs.isInt(), ctx,  error: "Expected int as 1st argument to " + op + "; actual type: " + lhs);
        this.check(rhs.isInt(), ctx,  error: "Expected int as 2nd argument to " + op + "; actual type: " + rhs);
        break;
}

switch (op) {
    // Only AND and less-than return booleans;
    // all other operations return ints.
    case "&&":
    case "||":
    case "<":
        this.types.push(new Type(Kind.BOOLEAN));
        break;
    default:
        this.types.push(new Type(Kind.INT));
        break;
```

2. For babycino to reject ill-formed || expressions, the || had to be included in the type checker. As || is a Boolean operator, babycino needed to check the type of the input. Therefore, the TypeChecker.java file was modified in order to make babycino only accept Boolean inputs for ||.

3. Below is the code for the MiniJava program to reject ill-formed || expressions. The program tests the || against an int and a Boolean parameter in three separate tests.

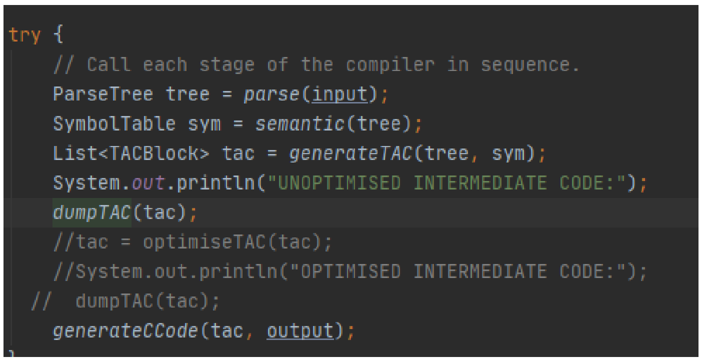


```
class SubtaskTwo{
    public static void main(String[] args){
        System.out.println(new SubtaskTwoTest().test( boolArg: true,  intArg: 5));
    }
}

class SubtaskTwoTest{
    public int test(boolean boolArg, int intArg){
        boolean resultOne ; //stores the result from the first test
        boolean resultTwo; //stores the result from the second test
        boolean resultThree; //stores the result from the third test

        resultOne = boolArg || intArg;
        resultTwo = intArg || boolArg;
        resultThree = intArg || intArg;

        return 0;
    }
}
```



```
C:\Users\RORY JACKSON\Documents\Uni-Year2\Compilers\babycino>java -cp .;
Expected boolean as 2nd argument to ||; actual type: int
Context: boolArg||intArg
Expected boolean as 1st argument to ||; actual type: int
Context: intArg||boolArg
Expected boolean as 1st argument to ||; actual type: int
Context: intArg||intArg
Expected boolean as 2nd argument to ||; actual type: int
Context: intArg||intArg
Exiting due to earlier error.
```

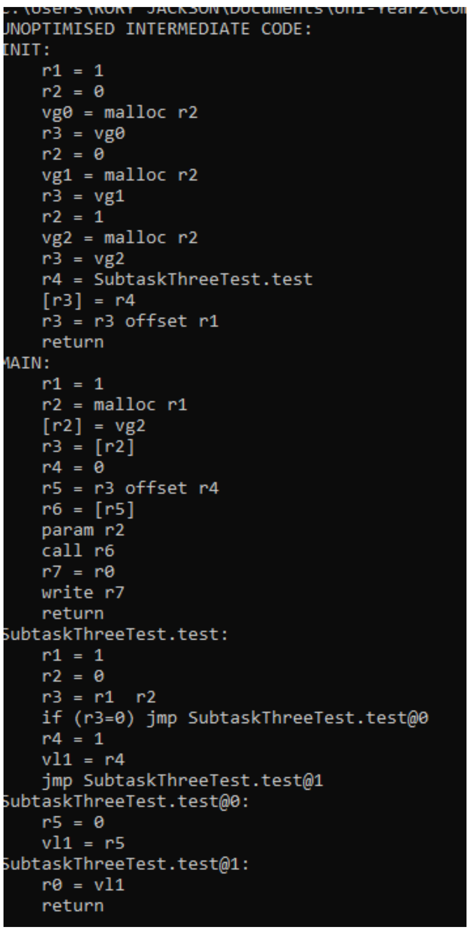
Subtask 3: Code Generation

1. 

```
try {
    // Call each stage of the compiler in sequence.
    ParseTree tree = parse(input);
    SymbolTable sym = semantic(tree);
    List<TACBlock> tac = generateTAC(tree, sym);
    System.out.println("UNOPTIMISED INTERMEDIATE CODE:");
    dumpTAC(tac);
    //tac = optimiseTAC(tac);
    //System.out.println("OPTIMISED INTERMEDIATE CODE:");
  //  dumpTAC(tac);
    generateCCode(tac, output);
```

2. In order to generate the unoptimized intermediate code, the file *babycino.java* was edited. The code that contained the "UNOPTIMISED INTERMEDIATE CODE" string was commented out, so it was uncommented along with *dumpTAC(tac).* Also *tac = optimiseTAC(tac)* is commented out as therefore, this saves the unoptimized code to a .c file.

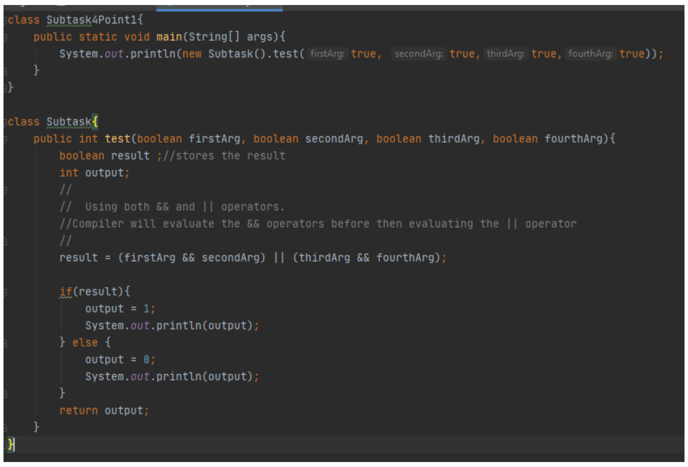
3. Below is a screenshot of the unoptimized intermediate code.



```
UNOPTIMISED INTERMEDIATE CODE:
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 = SubtaskThreeTest.test
    [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
SubtaskThreeTest.test:
    r1 = 1
    r2 = 0
    r3 = r1  r2
    if (r3=0) jmp SubtaskThreeTest.test@0
    r4 = 1
    vl1 = r4
    jmp SubtaskThreeTest.test@1
SubtaskThreeTest.test@0:
    r5 = 0
    vl1 = r5
SubtaskThreeTest.test@1:
    r0 = vl1
    return
```

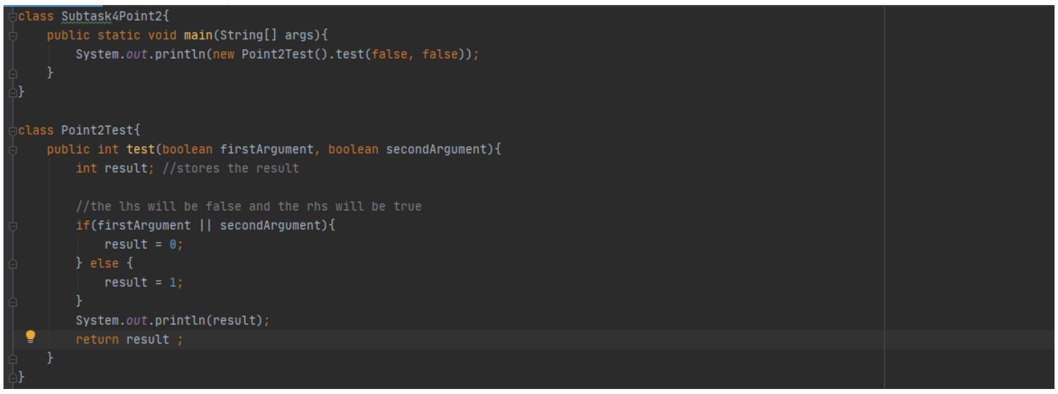
Subtask 4 Testing.

1. Below is the screenshot for testing point one. As && has higher precedence than || then the output should be 1. If the output did not satisfy the specification, then the output would be 0. Below is the screenshot for testing the point one. As && has higher precedence than || then the output should be 1. If the output did not satisfy the specification, then the output would be 0.



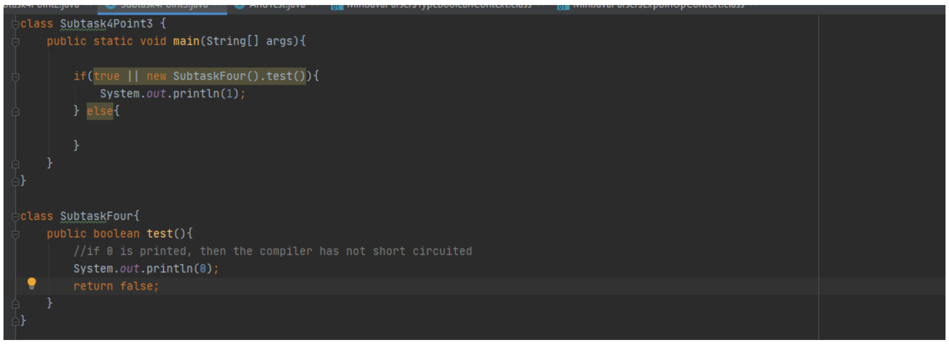
```
class Subtask4Point1{
    public static void main(String[] args){
        System.out.println(new Subtask().test( firstArg: true,  secondArg: true, thirdArg: true, fourthArg: true));
    }
}

class Subtask{
    public int test(boolean firstArg, boolean secondArg, boolean thirdArg, boolean fourthArg){
        boolean result ;//stores the result
        int output;
        //
        //  Using both && and || operators.
        //Compiler will evaluate the && operators before then evaluating the || operator
        //
        result = (firstArg && secondArg) || (thirdArg && fourthArg);

        if(result){
            output = 1;
            System.out.println(output);
        } else {
            output = 0;
            System.out.println(output);
        }
        return output;
    }
}
```

2. Below is the screenshot for testing point two. Both the left and the right operand are false. Therefore, the output will be false.



```
class Subtask4Point2{
    public static void main(String[] args){
        System.out.println(new Point2Test().test(false, false));
    }
}

class Point2Test{
    public int test(boolean firstArgument, boolean secondArgument){
        int result; //stores the result

        //the lhs will be false and the rhs will be true
        if(firstArgument || secondArgument){
            result = 0;
        } else {
            result = 1;
        }
        System.out.println(result);
        return result ;
    }
}
```

3. Below is the screen shot for testing point 3 and 4. As the first operand is true, all possible outcomes will be true. Therefore, babycino will not evaluate the right-hand side regardless of whether it is true or false. This also proves point 4.



```
class Subtask4Point3 {
    public static void main(String[] args){

        if(true || new SubtaskFour().test()){
            System.out.println(1);
        } else{

        }
    }
}

class SubtaskFour{
    public boolean test(){
        //if 0 is printed, then the compiler has not short circuited
        System.out.println(0);
        return false;
    }
}
```