University of Reading
School of Mathematical, Physical and Computational Sciences
CS2CO16-Compilers
Stefanos Stefanou
27020363

Subtask 1: Parsing(20%)

To be able to parse OR operations, we need to slightly modify our MiniJava.g4 file, as follows.

```
41      | expression ( '<' ) expression                    # ExpBinOp
42      | expression ( '&&' ) expression                   # ExpBinOp
43      | expression ( '||' ) expression                   # ExpBinOp
44      | INT         # ExpConstInt
45      | 'true'      # ExpConstTrue
46      | 'false'     # ExpConstFalse
```

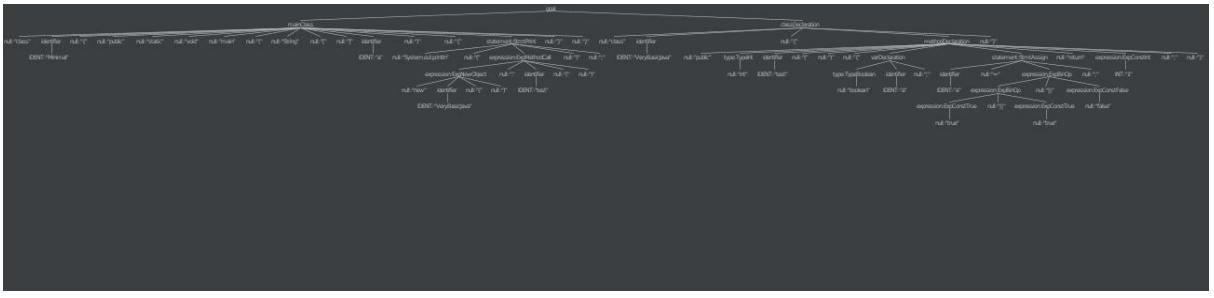
We added an additional rule to our parser, the change makes our parser capable of recognising expressions of the form

Expr || Expr

Below a minimal java program with its parse tree given, the Image was created using the ANTLRv4-intellij-plugin for Intellij IDEA IDE

```
class Minimal{
    public static void main(String[] a) { System.out.println(new VeryBasicJava().test()); }
}
class VeryBasicJava {
    public int test(){
        boolean a;
        a=true||true||false;
        return 1;
    }
}
```

This code with the generated .png are available at {Project}/subtasks/task1



Subtask 2 : Semantic analysis(20%)

We need to modify our TypeChecker, now we have a second type who takes booleans, so we need to make sure that the expressions at the left-hand side and the right-hand side are always booleans
TypeChecker.java

```
147        switch (op) {
148            // AND is the only operator that takes booleans, not ints.
149            case "&&":
150            case "||":
151                this.check(lhs.isBoolean(), ctx,  error: "Expected boolean as 1st argument to &&; actual type: " + lhs);
152                this.check(rhs.isBoolean(), ctx,  error: "Expected boolean as 2nd argument to &&; actual type: " + rhs);
153                break;
154            default:
155                this.check(lhs.isInt(), ctx,  error: "Expected int as 1st argument to " + op + "; actual type: " + lhs);
156                this.check(rhs.isInt(), ctx,  error: "Expected int as 2nd argument to " + op + "; actual type: " + rhs);
157                break;
158        }
```

The result of an OR binary operation is always Boolean, so we need to push a new BOOLEAN type into our types local stack
TypeChecker.java

```
159
160            switch (op) {
161                // Only AND and less-than return booleans;
162                // all other operations return ints.
163                case "&&":
164                case "||":
165                case "<":
166                    this.types.push(new Type(Kind.BOOLEAN));
167                    break;
168                default:
169                    this.types.push(new Type(Kind.INT));
170                    break;
171            }
172        }
```

The program below fails in the semantic analysis phase, due to invalid types involved into an OR BinOp

```
class SematicallyErroneousJavaMain{
    public static void main(String[] a) { System.out.println(new SematicallyErroneousJava().test()); }
}
class SematicallyErroneousJava {
    public int test(){
        boolean a;
        int b;
        b=10;
        a=true||b;
        return 1;
    }
}
```

Subtask 3 ,code generation(20%)

For the code generation to take place, we need to modify our TACGenerator to create the intermediate code for the ‘||” (OR) case. The problem is that our BinOp is not directly supported by the Three Address Code, as a result, we need to implement in terms of the supported types. Given that the boolean type is just an integer in TAC-Level (0:false,1:true), my algorithm here is very simple

1. Tranfer the first expression result to a register
2. Subtract 1 from this register
3. Short-circuit if the result is 0 (if the result is 0 then it was true, and no further actions need to be taken).

README -> due to one-page-limit , i will test for points 1,2,3 and 4 in the testing section.

```
246         else if(    op.equals("||")){
247             // || should short-circuit.
248             String expr1fail = this.genlab();              //jump label : short-circut if expression1 is true
249             String expr2fail = this.genlab();              //jump label : short-circut if expression2 is true
250             String end = this.genlab();                    //the final label
251             String curr = this.genreg();                   //A temporary register for our convenience
252             String mid = this.genreg();                    //we subtract -1 in two places , so we load this register with -1
253             String zero = this.genreg();                   //this register is always zero ()
254             String finalresult = this.genreg();            //this register will hold the final value
255             result.addAll(expr1);                          //add the left expression in
256
257
258             //evaluate the first expression , if is true then short circut(jump)
259             result.add(TACOp.mov(curr, expr1.getResult()));
260             result.add(TACOp.immed(mid,  n: -1));
261             result.add(TACOp.immed(zero,  n: 0));
262             result.add(TACOp.add(curr,curr,mid));
263             result.add(TACOp.jz(curr, expr1fail));//short
264
265             //do the same at the second expression
266             result.addAll(expr2);
267             result.add(TACOp.mov(curr, expr2.getResult()));
268             result.add(TACOp.add(curr,curr,mid));
269             result.add(TACOp.jz(curr, expr2fail));//short
270
271             //both are false case
272             result.add(TACOp.mov(finalresult, expr1.getResult()));//both false
273             result.add(TACOp.jz(zero, end));//to end
274
275             //second is true  case
276             result.add(TACOp.label(expr2fail));
277             result.add(TACOp.mov(finalresult, expr2.getResult()));//2 failed
278             result.add(TACOp.jz(zero, end));//short
279             //first is true case
280             result.add(TACOp.label(expr1fail));
281             result.add(TACOp.mov(finalresult, expr1.getResult()));//1 failed
282             result.add(TACOp.jz(zero, end));
283             //End
284             result.add(TACOp.label(end));
285             result.setResult(finalresult);
286             return result;
287         }
```

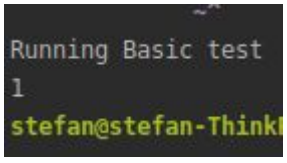
You can find the intermediate code at {PROJECT}/subtasks/task3

```
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 = VeryBasicJava.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
VeryBasicJava.test:
    r1 = 1
    r3 = r1
    r4 = -1
    r5 = 0
    r3 = r3 + r4
    if (r3=0) jmp VeryBasicJava.test@0
    r2 = 1
    r3 = r2
    r3 = r3 + r4
    if (r3=0) jmp VeryBasicJava.test@1
    r6 = r1
    if (r5=0) jmp VeryBasicJava.test@2
VeryBasicJava.test@1:
    r6 = r2
    if (r5=0) jmp VeryBasicJava.test@2
VeryBasicJava.test@0:
    r6 = r1
    if (r5=0) jmp VeryBasicJava.test@2
VeryBasicJava.test@2:
    r8 = r6
    r9 = -1
    r10 = 0
    r8 = r8 + r9
    if (r8=0) jmp VeryBasicJava.test@3
    r7 = 0
    r8 = r7
    r8 = r8 + r9
    if (r8=0) jmp VeryBasicJava.test@4
    r11 = r6
    if (r10=0) jmp VeryBasicJava.test@5
VeryBasicJava.test@4:
    r11 = r7
    if (r10=0) jmp VeryBasicJava.test@5
VeryBasicJava.test@3:
    r11 = r6
    if (r10=0) jmp VeryBasicJava.test@5
VeryBasicJava.test@5:
    vl1 = r11
    r12 = 1
    r0 = r12
    return
```

Testing(20%)
The following tests cover all the points given in our coursework specification, below is a summarized table of all the tests i performed

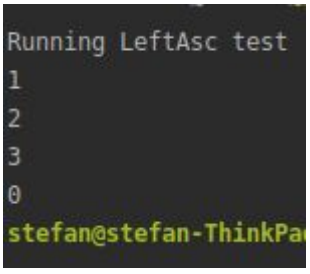
Class Name	Description	Status
BasicTest	Test every possible combination (the truth table)	Passed
LeftAsc	Left associativity	Passed
Short Circuit	Does Not evaluate the a||b if a is true	Passed
Precedence	Checks if || has lower precedence than &&	Passed

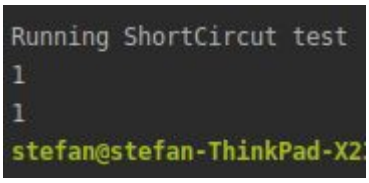
**The listings of the tests are in the last page.



Basic Test , just tests that every other combination other than false||false returns true , The definition of OR. This returns true as shown(true -> passed).

LeftAsc : This test ensures that the left-associativity rule will apply, under this setup.To pass the test, the results 1, 2, 3 needs to be printed on screen (showing the order of evaluation is left-associative). The following screenshot proves that



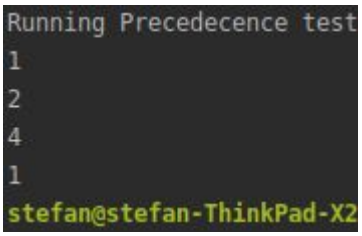


ShortCircut: This test proves that in case of an OR expression has the first element true, the rest of the expression automatically becomes true and the rest of the expression is not evaluated

Precedence : This is the more complex test and it needs clarification, let the following expression

false || false && false || true

If in the following expression we have the && operator with higher precedence than || and with left-associativity in mind, we expect this expression to evaluate to true
Alternatively , if the || has higher precedence than && , then this expression will evaluate to false
if || has priority then (1) false || false -> false (2) false || true -> true (3) false & true -> false
if && has priority then (1) false & false -> false (2) false|false -> false (3) false | true -> true
So if && priority -> then true and if || priority -> then false



This test evaluates to true , proving that && has higher precedence than ||
Some useful notes

1. The test code for each section can be found under {Project}/subtasks/
2. You can also find the C code and binaries under the {Project}/subtasks/task4
3. You can run these tests by running the customised .sh scripts who i made, named basicTest.sh,leftAsc.sh,sortCircut.sh,precedence.sh
4. Alternatively , you can run ./runAllTests.sh who will generate 4 reports for you ,one for each test. You can finally call ./clean.sh and ./cleanreports.sh for cleaning your environment