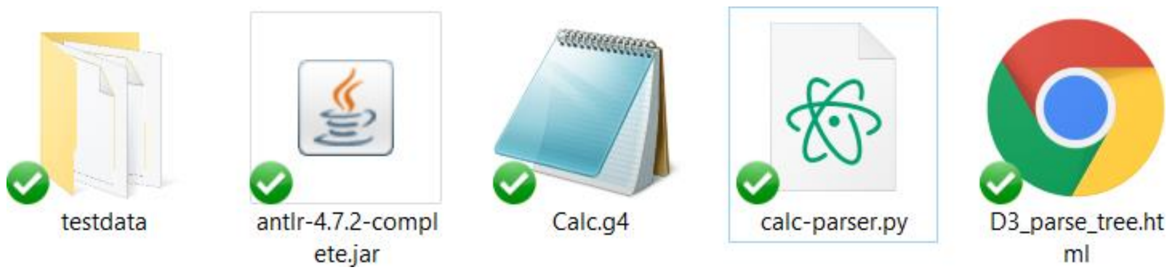
6G6Z1110 Programming Languages: Principles and Design
Dr Ryan Cunningham, E142, ryan.cunningham@mmu.ac.uk

Week 4 Exercises (Syntax Analysis with Antlr4 and Python3)

(Task 1) Writing Syntax Rules in Antlr4 (Warmup Task)

In this short warmup task, you will work with some starter files and develop a syntax analyser using Antlr4 and Python3. We are going to continue working on Calc (language introduced in week 2). Today we will work on the syntax analysis part, and in future weeks we will develop the semantic checking and code generation parts.

1.1 Download the **Cacl Parser Starter Files** under week 4 in this unit's Moodle area and unzip them to a suitable location. When you have unzipped them, you should see the following files:



The **testdata** folder contains some files to help you test your parser. The **antlr-4.7.2-complete.jar** is the language recognition tool Antlr4, which you will use to create a parser. The **Calc.g4** is a text file in which you will define the grammar for our language. Those files are the same as the files in the week 2 starter files folder.

There are 2 new files: The **calc-parser.py** is a Python script in which you will load the lexer and parser created by Antlr and process some text files in the **testdata** folder. The **D3_parse_tree.html** file is a D3.js based browser file which you will use to visualise your parse tree for debugging purposes etc.

Note: You should use the **Calc.g4** file you have already worked on in week 2.

1.1 Open a terminal and navigate to the folder containing your Calc.g4 and Antlr4 jar files. Then enter the following command:

```
java -Xmx500M -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool -Dlanguage=Python3 Calc.g4 -visitor
```

This command is the same as for last week. However, **notice the -visitor** option added to the end of the command. This tells Antlr to generate a visitor interface (more on that later), which is a **CalcVisitor.py** (see below) file that enables interaction with the parse tree generated after analysing some input text.

There should be no errors and a bunch of files should have been created for you – if there are, and you cannot resolve them, ask the tutor for help.

The following files should have been created for you:

Calc.interp
Calc.tokens
CalcLexer.py
CalcLexer.tokens
CalcListener.py
CalcParser.py
CalcVisitor.py
CalcLexer.interp

1.2 Open the calc-parser.py file in **Spyder** or **Atom** (or some other code/text editor).

```
 1  import antlr4 as ant
 2  from CalcLexer import CalcLexer
 3  from CalcParser import CalcParser
 4
 5  filein = open('testdata/test_01.calc', 'r')
 6  lexer = CalcLexer(ant.InputStream(filein.read()))
 7
 8  stream = ant.CommonTokenStream(lexer)
 9  parser = CalcParser(stream)
10  tree = parser.calculator()
```

On lines 1 – 2 we import the required Antlr library the lexer and parser we just created using the Antlr tool.

On line 5 we open one of the test files to use as our input source.

On line 6 we create a new lexer using our test file as input.

On line 8 create a new token stream to use as input to the parser.

On line 9 we create a new parser using the token stream as input.

On line 10 we attempt to parse the token stream using the **calculator** rule.

Execute this Python code and observe the output.

```
line 1:1 mismatched input 'x' expecting ')'
line 1:7 token recognition error at: '\n'
```

You will see that the parser reports an error (above). The message says that the parser saw an '**x**', but it was expecting to see a '**)**'.

Refer to the **test_01.calc** file:

```
(x + y)
```

Then refer to the **Calc.g4** file for the definition of the rule **calculator**:

```
calculator : LBRACE RBRACE;
```

Here we see the problem: The **calculator** rule is just an open brace and a closing brace, so if the parser finds any other symbol it will thrown an error. This is one of the benefits of parsing – error catching and reporting. Refer back to the calculator rule in the Calc grammar from week 2:

<calculator>	→	**(** <statement>* <expr> **)**

This rule accepts a left brace, a bunch of statements, and a single expression, followed by a closing brace.

The text in **test_01.calc** is a single expression ***x + y***, so we can get that file to parse correctly by defining the **expr** rule and adding that to the **calculator** rule.

```
calculator : LBRACE expr RBRACE;

expr : VAR
     | expr op expr;

op : ADD;
```

Note: all syntax rules are in lower case and use the same regular expression syntax as the lexer rules.

In the above code snippet, we have modified the **calculator** rule, adding a single **expr** between open and closing braces. Then partially defined the expr rule, which can be a single **VAR** (variable name defined in the lexer rules) or an infix op applied to two expressions. We have also partially defined **op** as just an **ADD** (plus symbol).

Edit your Calc.g4 file as above then enter the following command in a terminal to rebuild your parser:

```
java -Xmx500M -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool -Dlanguage=Python3 Calc.g4 -visitor
```

Now run your calc-parser.py code again. You should get no errors.

1.3 Complete the rest of the **expr** and **op** rules. Test your solutions using the supplied test files.

Note: not all test files can be parsed with just the **expr** and **op** rules defined.

(Task 2) Visualising the Parse Tree using the Tree Visitor (Warmup Task)

In the previous task you defined some rules for a partially valid calculator program. If you experienced writing incorrect rules, the parser would throw an error and you would have some idea of how to debug and fix your **Calc.g4** grammar. When successful, the parser would give no feedback, and you can only assume your grammar is correct. However, a grammar may parse a string but not have the intended result. We can debug such errors by rendering and viewing the resulting parse tree.

Antlr has a feature for printing parse trees, but that feature is not installed on the machine. This task is an opportunity to explore the **visitor** interface class to render the parse tree yourself. This task is a very good warmup task before **semantic analysis** and **code generation** labs, where you will use very similar techniques.

2.1 In Atom, open the **D3_parse_tree.html** file and observe lines 22 – 24.

```
21        <script>
22          var treeData = [
23            //****paste ANTLR-4 tree-visitor string here****
24          ]
```

On line 23 there is a comment asking you to paste a string from your parser here. If you open **D3_parse_tree.html** using a browser, currently it does nothing. In this task we will implement the visitor interface so that a string is printed, representing the parse tree, which will be rendered in the browser if successful.

In calc-parser.py, add lines 4 – 9 as highlighted in the code below.

```
 1  import antlr4 as ant
 2  from CalcLexer import CalcLexer
 3  from CalcParser import CalcParser
 4  from CalcVisitor import CalcVisitor
 5
 6  class CalcPrintVisitor(CalcVisitor):
 7      def __init__(self):
 8          super().__init__()
 9          self.text = ''
10
11  filein = open('testdata/test_01.calc', 'r')
12  lexer = CalcLexer(ant.InputStream(filein.read()))
13
14  stream = ant.CommonTokenStream(lexer)
15  parser = CalcParser(stream)
16  tree = parser.calculator()
```

Line 4 imports the visitor interface generated by Antlr.

Lines 6 – 9 create a new visitor class which inherits from that visitor interface and implements a constructor which creates an empty string **text**.

The visitor class, walks the parse tree one node at a time in a depth-first manner, calling defined functions representing each terminal or nonterminal symbol in the tree. Note that this is after parsing and a valid parse tree has been created (line 16).

Add the following lines of code (at the bottom of your .py file) to get Antlr to walk the tree using your visitor class:

```python
visitor = CalcPrintVisitor()
visitor.visit(tree)
print(visitor.text)
```

If you run the code, you will notice no change, because we have not implemented any of the methods in the visitor class.

2.2 Open the **CalcVisitor.py** file generated by Antlr and observe the contents.

```python
class CalcVisitor(ParseTreeVisitor):

    # Visit a parse tree produced by CalcParser#calculator.
    def visitCalculator(self, ctx:CalcParser.CalculatorContext):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by CalcParser#expr.
    def visitExpr(self, ctx:CalcParser.ExprContext):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by CalcParser#op.
    def visitOp(self, ctx:CalcParser.OpContext):
        return self.visitChildren(ctx)
```

You should see something very similar to the above snippet. Within the **CalcVisitor** class, there are 3 methods each following a naming convention: **visit<rule_name>(ctx)**. There is one method for each syntax rule defined in your **Calc.g4** file. In this case, one for **calculator**, one for **expr**, and one for **op**. As Antlr walks your parse tree it will invoke these methods in the order they are visited. We can use this pattern to generate text to render a parse tree by override these methods.

Copy and paste these methods into your **calc-parser.py** code (see highlight below).

```python
class CalcPrintVisitor(CalcVisitor):
    def __init__(self):
        super().__init__()
        self.text = ''

    def visitCalculator(self, ctx:CalcParser.CalculatorContext):
        return self.visitChildren(ctx)

    def visitExpr(self, ctx:CalcParser.ExprContext):
        return self.visitChildren(ctx)

    def visitOp(self, ctx:CalcParser.OpContext):
        return self.visitChildren(ctx)
```

Now add these **printNode** and **visitTerminal** methods just below the constructor a simple copy-paste should work:

```
def printNode(self, ctx):
    self.text = self.text + '{\"name\": \"' + parser.ruleNames[ctx.getRuleIndex()] + '\",' + '\"value\":2,' + '\"children\":['
    self.visitChildren(ctx)
    self.text = self.text + ']},'

def visitTerminal(self, ctx):
    self.text = self.text + '{\"name\": \"' + ctx.getText().replace('\\', '\\\\').replace('\"', '\\\"') + '\"},'
```

Finally, replace the body of the visitCalculator, visitExpr, and visitOp methods with:

```
self.printNode(ctx)
```

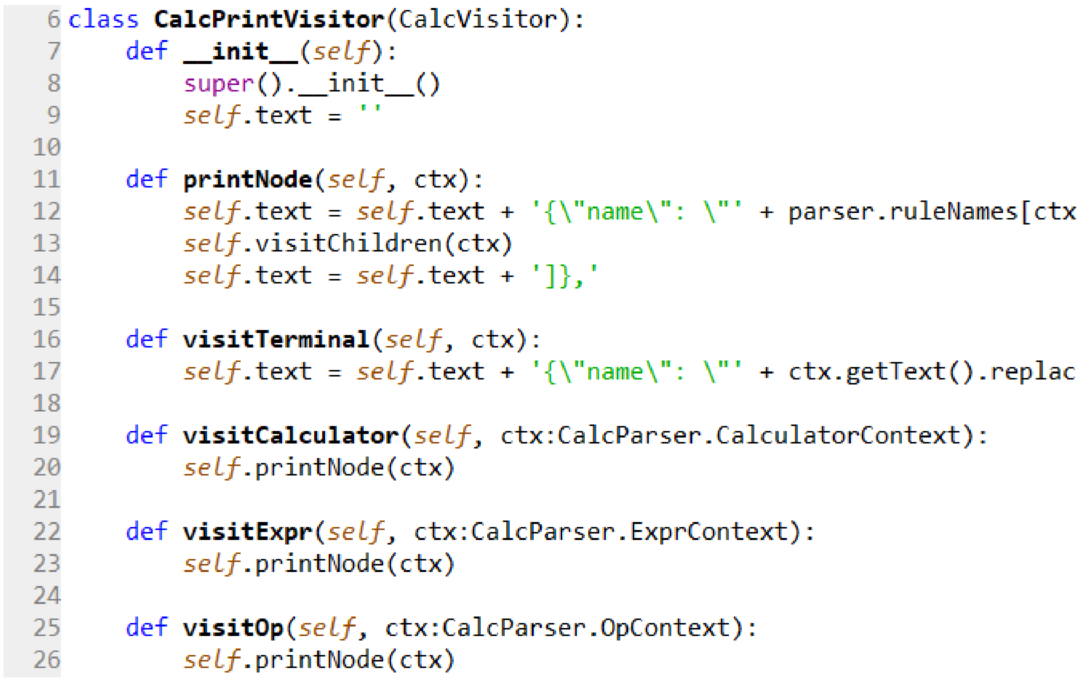
i.e. remove the return **self.visitChildren(ctx)** and write the above code in its place.

Run the code again.

```
{"name": "calculator","value":2,"children":[{"name": "("},{"name": "expr","value":2,"children":
[{"name": "expr","value":2,"children":[{"name": "x"},]},{"name": "op","value":2,"children":[{"name":
"+"},]},{"name": "expr","value":2,"children":[{"name": "y"},]},]},{"name": ")"},]},
```

You should see something similar to that printed above. If not, and you cannot resolve it, ask the tutor for help.

For reference, my code looks like this when all put together:

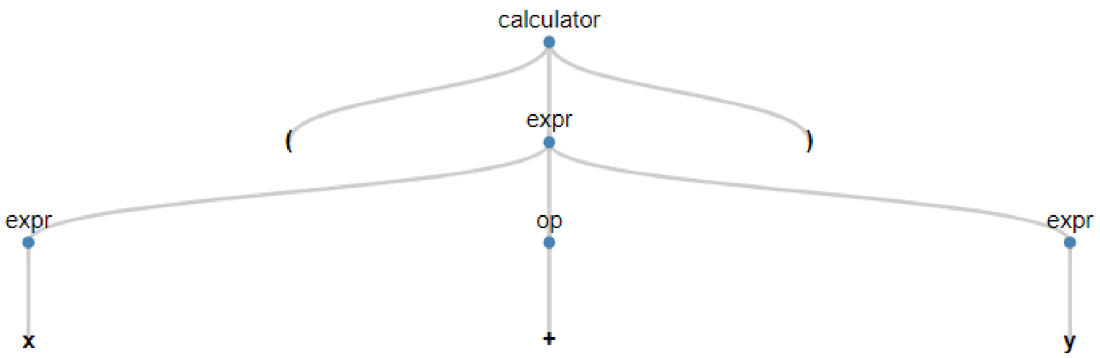


```
class CalcPrintVisitor(CalcVisitor):
    def __init__(self):
        super().__init__()
        self.text = ''

    def printNode(self, ctx):
        self.text = self.text + '{\"name\": \"' + parser.ruleNames[ctx
        self.visitChildren(ctx)
        self.text = self.text + ']},'

    def visitTerminal(self, ctx):
        self.text = self.text + '{\"name\": \"' + ctx.getText().replac

    def visitCalculator(self, ctx:CalcParser.CalculatorContext):
        self.printNode(ctx)

    def visitExpr(self, ctx:CalcParser.ExprContext):
        self.printNode(ctx)

    def visitOp(self, ctx:CalcParser.OpContext):
        self.printNode(ctx)
```

2.3 In Atom, open the **D3_parse_tree.html** file and copy the text printed by your visitor class, then paste it inside the tree data, where the comment asks you to.

```
        var treeData = [
          //****paste ANTLR-4 tree-visitor string here****
          {"name": "calculator","value":2,"children":[{"name": "("
        ]
```

Your **D3_parse_tree.html** should now look like this. Save it, then open it in a browser.



You should see a nicely formatted parse tree for the **test_01.calc** program (x + y). If not, and you can't resolve it, ask a tutor for help.

(Task 3) Rule Precedence in Antlr4

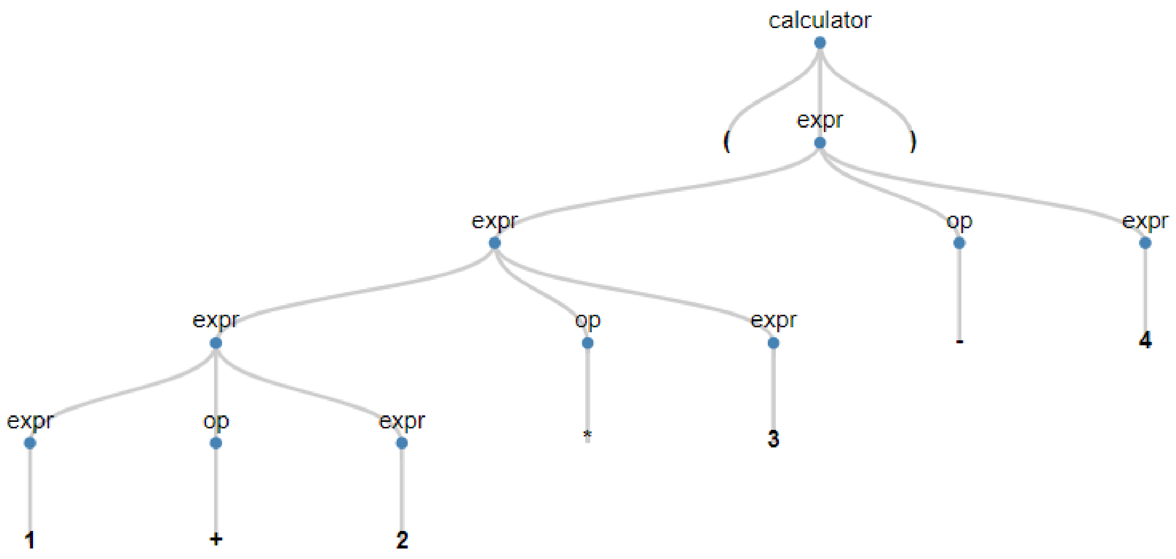
Now that you have experienced writing syntax rules in Antlr4, and can visualise a parse tree for a valid sentence in the defined language, you will see how you can use the parse tree to debug your grammar. One of the situations where visualising the parse is necessary is rule precedence (e.g. operator precedence). Rules with higher precedence should appear deeper in the tree (see the lecture and reading material for more details).

For the Calc language, we will define the following order of precedence for operations in expressions:

Operator	Description
*, /	Multiplication, division
+, -	Addition, subtraction

Normally, as per the lecture, you would have to manually rewrite the grammar to remove the ambiguity such that only 1 parse tree is possible, and that parse tree encodes the order of precedence of each operation as per the table above. As with most language tools, Antlr provides a mechanism for encoding the order of precedence in the rule definitions.

3.1 Generate and visualise the parse tree for the **test_06.calc** file.



If you have implemented the required rules you will find that the parse tree generated describes an arbitrary order of precedence where the left-most symbol has the highest precedence. To fix it, all we have to do is define the rules in the order of their precedence, grouping those with the same level of precedence:

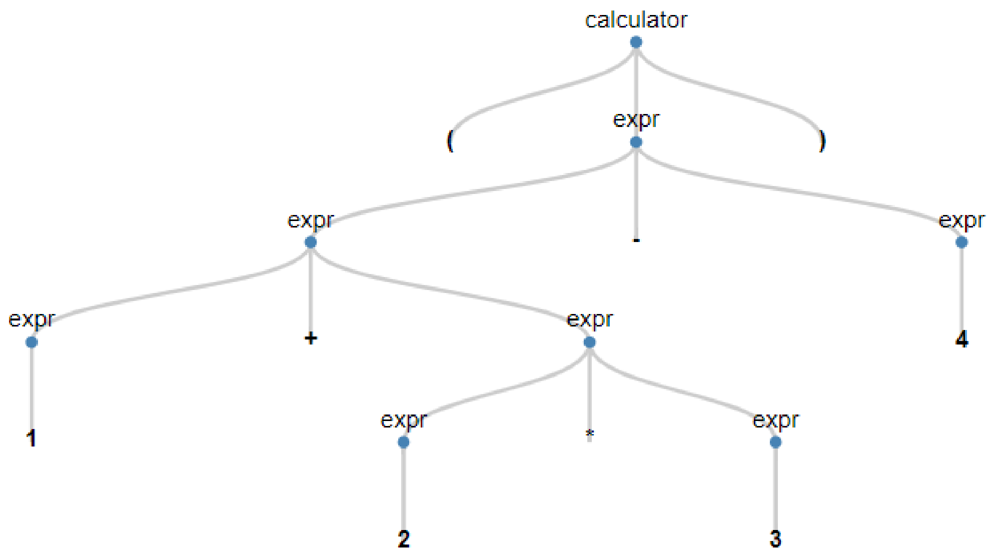
For example:

expr : expr (MUL | DIV) expr

| expr (ADD | SUB) expr;

Notice that we have removed the op rule. Antlr will only encode precedence at the highest level of a non-terminal, so we have to factor out the op rule.

Modify your grammar in order to produce the following parse tree:



Above you can see that the multiply operation lies in the deepest part of the tree, which adheres to the operator precedence order defined in the precedence table above.

Tip: You will likely find the following error:

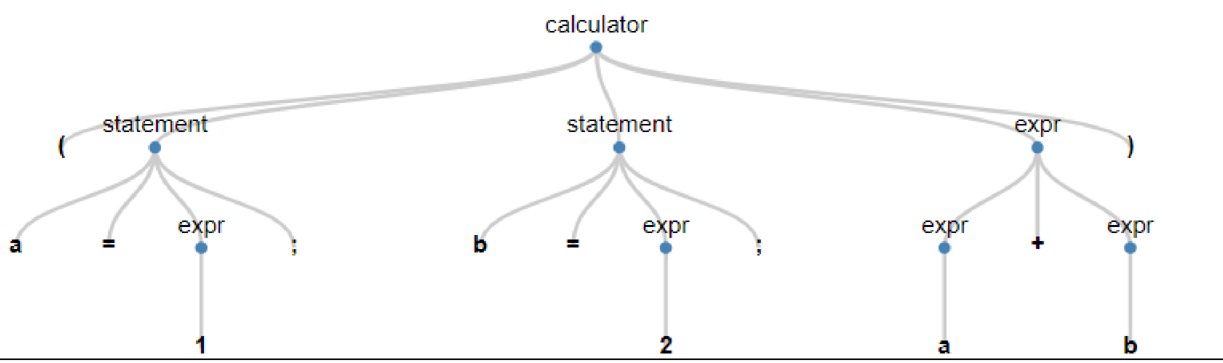
```
AttributeError: type object 'CalcParser' has no attribute 'OpContext'
```

That is because you have removed the op rule, so your visitor class no longer matches the rules in the grammar. Simply remove the **visitOp** method from your **CalcPrintVisitor** class. Likewise, when you add more rules, you need to add the methods to your **CalcPrintVisitor** class, else they will not appear in your parse tree.

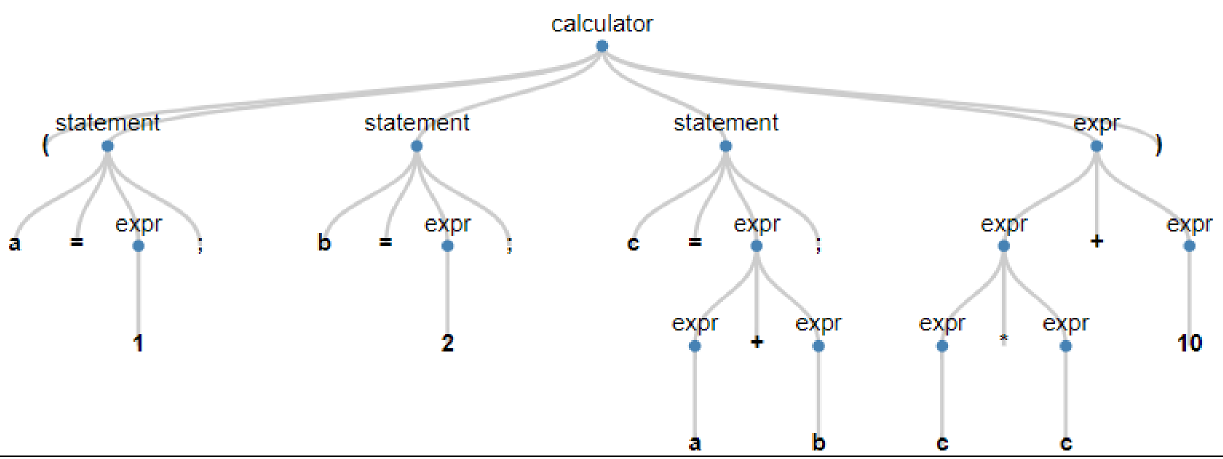
(Task 4) Calc Syntax Rules

Now that you are confident with writing syntax rules using Antlr. write the remaining grammar for the Calc language. Check your solution against the expected solutions for a couple of selected input files shown below.

test_04.calc



test_05.calc

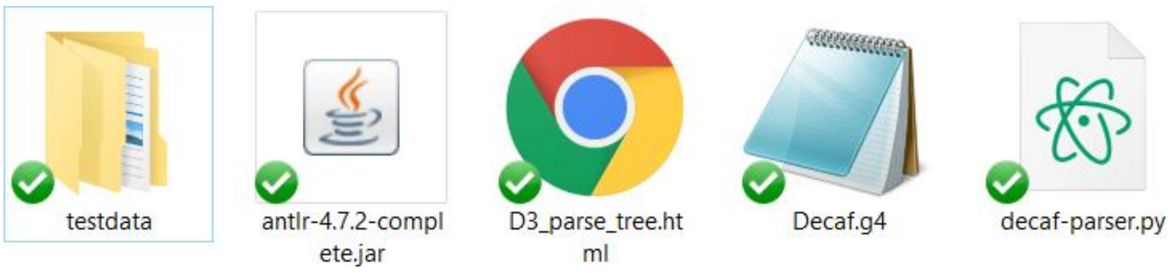


(Task 5) Decaf Parser (Coursework)

Now that you have practiced writing parser rules for the simple Calc language, you should **start work on your coursework**. Please do ask for help from the tutors, but **do not expect answers to be given** – your coursework is a formal piece of work which is to be worked on all term – some of what you do not understand right now will become clear in future weeks.

In this task, start to write your parser rules, referring to the Decaf language specification document on Moodle.

5.1 Download the **Decaf Parser Starter Files** from this unit's Moodle area (week 4 – (Term 2) Compilers) and unzip them to a suitable location. When you have unzipped them, you should see the following files:



Open **decaf-parser.py** and observe the contents. You will notice that the visitor class has been started for you. As you add parser rules to your **Decaf.g4** files, you will need to keep that class up to date so you can print appropriate parse tree for debugging etc.

Note: You should replace **Decaf.g4** with the **Decaf.g4** you already started working on in week 2.

Observe line 27:

```
25 stream = ant.CommonTokenStream(lexer)
26 parser = DecafParser(stream)
27 tree = parser.program()
```

Because the **Decaf** grammar is quite a bit more complicated than the **Calc** grammar, you will probably want to test different parts of your parser rules, rather than the root rule **program**.

For example, if you are working on expressions, and you have defined a rule called **expr**, you can create your own text file containing only an expression, then change line 27 to parse with **expr** as the root node of the tree rather than program. For example:

Source text:

a + b * c

decaf-parser.py:

```
25 stream = ant.CommonTokenStream(lexer)
26 parser = DecafParser(stream)
27 tree = parser.expr()
```

Note line 27 has been changed from **program()** to **expr()**.