

Lab 8 - Abstract Syntax Tree and Code Runner - Report

Students:

1. Phan Trần Thanh Huy – ITCSIU22056
-

1 Overview

In this section, briefly listing the issues that the class today covers.

2 Results

Q.1. Trace the process of generating an AST for the input expression $3 + 5 * 2$. Check the resulting AST with your tracing in the given code. Sketch the workflow of your trace from input to the output AST (i.e., step-by-step).

Step 1: Run the command “python run.py test” to generate.

Step 2: Get the text from testcase.txt

For example: “3 + 5 * 2”

Step 3: Running the Sample.g4 to create the rule before testing.

Step 4: It will separate the text into token

```

32     Add = 1
33     Sub = 2
34     Mul = 3
35     Div = 4
36     Integer = 5
37     WS = 6
38
39     channelNames = [ u"DEFAULT_TOKEN_CHANNEL", u"HIDDEN" ]
40
41     modeNames = [ "DEFAULT_MODE" ]
42
43     literalNames = [ "<INVALID>",
44 |         "'+'", "'-'", "'*'", "'/'" ]
45
46     symbolicNames = [ "<INVALID>",
47 |         "Add", "Sub", "Mul", "Div", "Integer", "WS" ]
48
49     ruleNames = [ "Add", "Sub", "Mul", "Div", "Integer", "WS" ]
50
51     grammarFileName = "Sample.g4"

```

, check token syntax (skip off-channel tokens)

```

47     def LB(self, k:int):
48         if k==0 or (self.index-k)<0:
49             return None
50         i = self.index
51         n = 1
52         # find k good tokens looking backwards
53         while n <= k:
54             # skip off-channel tokens
55             i = self.previousTokenOnChannel(i - 1, self.channel)
56             n += 1
57         if i < 0:
58             return None
59         return self.tokens[i]

```

, check grammar and sentence syntax.

```

61     def LT(self, k:int):
62         self.lazyInit()
63         if k == 0:
64             return None
65         if k < 0:
66             return self.LB(-k)
67         i = self.index
68         n = 1 # we know tokens[pos] is a good one
69         # find k good tokens
70         while n < k:
71             # skip off-channel tokens, but make sure to not look past EOF
72             if self.sync(i + 1):
73                 i = self.nextTokenOnChannel(i + 1, self.channel)
74                 n += 1
75         return self.tokens[i]

```

```

28     grammarFileName = "Sample.g4"
29
30     atn = ATNDeserializer().deserialize(serializedATN())
31
32     decisionsToDFA = [ DFA(ds, i) for i, ds in enumerate(atn.decisionToState) ]
33
34     sharedContextCache = PredictionContextCache()
35
36     literalNames = [ "<INVALID>", "'+'", "'-'", "'*'", "'/'" ]
37
38     symbolicNames = [ "<INVALID>", "Add", "Sub", "Mul", "Div", "Integer",
39                       "WS" ]
40
41     RULE_program = 0
42     RULE_expression = 1
43     RULE_term = 2
44     RULE_factor = 3
45
46     ruleNames = [ "program", "expression", "term", "factor" ]
47
48     EOF = Token.EOF
49     Add=1
50     Sub=2
51     Mul=3
52     Div=4
53     Integer=5
54     WS=6

```

Then returning the parser. The general flow is shown below:

```

65     input_stream = FileStream(inputFile)
66     lexer = SampleLexer(input_stream)
67     stream = CommonTokenStream(lexer)
68     parser = SampleParser(stream)

```

Step 5: Based on the parser, it will analyze and generate the parse tree.

```

69     tree = parser.program()

```

Step 6: Based on the parse tree, generate the abstract syntax tree (AST) by ASTGeneration library.

```
72     from ASTGeneration import ASTGeneration
73     ast_generator = ASTGeneration()
74     asttree = tree.accept(ast_generator)
75     print('This is ast string: ', asttree)
76
77     runCode(asttree)
```

Step 7: Based on the AST, we will do the logic calculations and return the result from the CodeRunner file:

```
25     def runCode(astTree):
26         from CodeRunner import CodeRunner
27         code_runner = CodeRunner()
28         result = astTree.accept(code_runner)
29
30         print("Result:", result)
```

```
4     class CodeRunner():
5         def visitProgram(self, ctx:Prog):
6             return "\n".join([str(expr.accept(self)) for expr in ctx.expr])
7
8         def visitBinaryOp(self, ctx:BinOp):
9             left = ctx.left.accept(self)
10            right = ctx.right.accept(self)
11            if ctx.op == "+":
12                return left + right
13            elif ctx.op == "-":
14                return left - right
15            elif ctx.op == "*":
16                return left * right
17            elif ctx.op == "/":
18                return left / right
19
20        def visitInteger(self, node:Int):
21            return node.value
```

The result will show:

```
PS E:\Homework\PPL\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("+",Int(3),BinOp("*",Int(5),Int(2))))
Result: 13
```

Q.2. From Q.1, explore and explain how the grammar ensures that operators * and/ have higher precedence than + and- operators?

The code below:

```
10     def visitExpression(self, ctx:SampleParser.ExpressionContext):
11         if ctx.expression():
12             sign = ""
13             if ctx.Add():
14                 sign = "+"
15             elif ctx.Sub():
16                 sign = "-"
17
18         return BinOp(sign, ctx.expression().accept(self), ctx.term().accept(self))
19     else:
20         return ctx.term().accept(self)
21
22     def visitTerm(self, ctx:SampleParser.TermContext):
23         if ctx.term():
24             sign = ""
25             if ctx.Mul():
26                 sign = "*"
27             elif ctx.Div():
28                 sign = "/"
29
30         return BinOp(sign, ctx.term().accept(self), ctx.factor().accept(self))
31     else:
32         return ctx.factor().accept(self)
```

Overall explanation:

Step 1: The flow will run and check whether an expression or a term base on the operations. The expression will be "+", "-", and the term will be "*", "/".

The BinOp is the function which will display the AST.

```
30 @dataclass
31 class BinOp(Exp):
32     op:str
33     left:Exp
34     right:Exp
35
36     def __str__(self):
37         return "BinOp(\"" + self.op + "\", " + str(self.left) + ", " + str(self.right) + ")"
38
39     def accept(self, v):
40         return v.visitBinaryOp(self)
```

So when it is an expression, the BinOp will display the sign, the number and its term if there are any terms. Otherwise, it will display the term. Likewise, the term will display the sign and 2 numbers inside it. If it is only factor, it will continue run the visitFactor, or integer in the tree.

```

34  def visitFactor(self, ctx:SampleParser.FactorContext):
35      if ctx.Integer():
36          return self.visitInteger(ctx.Integer())
37
38  def visitInteger(self, node:SampleParser.Integer):
39      return Int(int(node.getText()))

```

The result:

```
This is ast string: Prog(BinOp("+",Int(3),BinOp("*",Int(5),Int(2))))
```

Moreover, we will have a test below:

This is the initial code:

```

10  def visitExpression(self, ctx:SampleParser.ExpressionContext):
11      if ctx.expression():
12          sign = ""
13          if ctx.Add():
14              sign = "+"
15          elif ctx.Sub():
16              sign = "-"
17
18          return BinOp(sign, ctx.expression().accept(self), ctx.term().accept(self))
19      else:
20          return ctx.term().accept(self)
21
22  def visitTerm(self, ctx:SampleParser.TermContext):
23      if ctx.term():
24          sign = ""
25          if ctx.Mul():
26              sign = "*"
27          elif ctx.Div():
28              sign = "/"

```

I will swap the mul and div to the expression, add and sub to the term below:

```
10     def visitExpression(self, ctx:SampleParser.ExpressionContext):
11         if ctx.expression():
12             sign = ""
13             if ctx.Mul():
14                 sign = "*"
15             elif ctx.Div():
16                 sign = "/"
17
18             return BinOp(sign, ctx.expression().accept(self), ctx.term().accept(self))
19         else:
20             return ctx.term().accept(self)
21
22     def visitTerm(self, ctx:SampleParser.TermContext):
23         if ctx.term():
24             sign = ""
25             if ctx.Add():
26                 sign = "+"
27             elif ctx.Sub():
28                 sign = "-"
29
30             return BinOp(sign, ctx.term().accept(self), ctx.factor().accept(self))
31         else:
32             return ctx.factor().accept(self)
```

Technically, swapping the operations of expression and term. On the other words, it will prior the "+", "-" first, then is "*", "/"

So the result:

```
This is ast string: Prog(BinOp("*",BinOp("+",Int(3),Int(5)),Int(2)))
Result: 16
```

Conclusion: The explanation is satisfied.

Q.3. Extend the code in Q.1 to include the following scenarios. For each, test with 3 different input samples. Capture the code you made modification to handle the problem, input samples and corresponding output.

Step 1: Add modulo and exponentiation into Sample.g4

```
1  grammar Sample;
2
3  program: expression;
4
5  expression: expression (Add | Sub) term | term;
6
7  term: term (Mul | Div | Mod | Exp) factor | factor;
8
9  factor: Integer;
10
11 Add : '+';
12 Sub : '-';
13
14 Mul : '*';
15 Div : '/';
16 Mod : '%';
17 Exp : '^';
18
19 Integer: [0-9]+ ;
20
21 WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines
```

Also run python run.py gen to generate lexer and parser.

Step 2: Because it is modulo and exponentiation, it will be term. Modifying the visitTerm in ASTGeneration.py

```
22     def visitTerm(self, ctx:SampleParser.TermContext):
23         if ctx.term():
24             sign = ""
25             if ctx.Mul():
26                 sign = "*"
27             elif ctx.Div():
28                 sign = "/"
29             elif ctx.Mod():
30                 sign = "%"
31             elif ctx.Exp():
32                 sign = "^"
```

Step 3: Add the logic calculation in visitBinaryOp inside CodeRunner.py


```

8      def visitBinaryOp(self, ctx:BinOp):
9          left = ctx.left.accept(self)
10         right = ctx.right.accept(self)
11         if ctx.op == "+":
12             return left + right
13         elif ctx.op == "-":
14             return left - right
15         elif ctx.op == "*":
16             return left * right
17         elif ctx.op == "/":
18             return left / right
19         elif ctx.op == "%":
20             return left % right
21         elif ctx.op == "^":
22             return left ** right

```

Q.3.1. Scenario 1: include the modulo operator (%).

Sample input: 10 % 3.

Result:

```

1  10 % 3

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL POSTMAN CONSOLE PORTS

```

PS E:\Homework\PP1\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("%",Int(10),Int(3)))
Result: 1

```

Sample input: 10 % 3 + 5 * 2.

Result:

```

tests > testcase.txt
1  10 % 3 + 5 * 2

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL POSTMAN CONSOLE PORTS

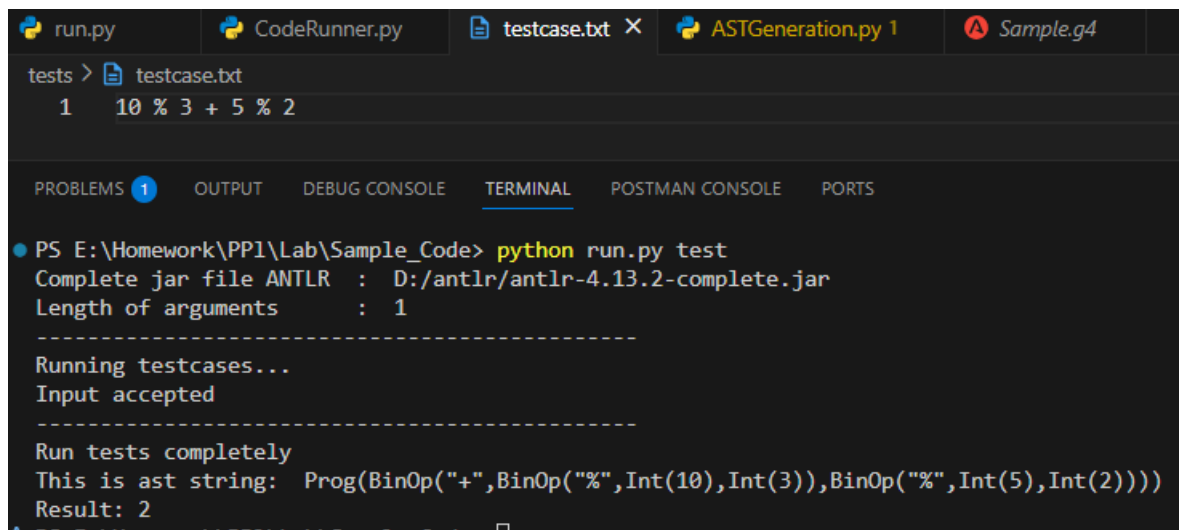
```

PS E:\Homework\PP1\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("+",BinOp("%",Int(10),Int(3)),BinOp("*",Int(5),Int(2))))
Result: 11
PS E:\Homework\PP1\Lab\Sample_Code>

```

Sample input: $10 \% 3 + 5 \% 2$.

Result:



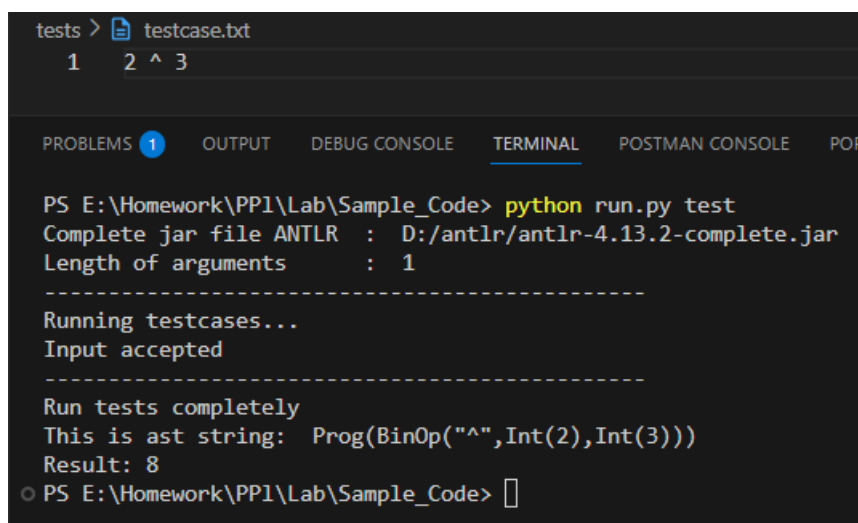
The screenshot shows an IDE with several tabs: `run.py`, `CodeRunner.py`, `testcase.txt`, `ASTGeneration.py 1`, and `Sample.g4`. The `testcase.txt` tab is active, showing the input `1 10 % 3 + 5 % 2`. The `TERMINAL` tab is also active, displaying the following output:

```
PS E:\Homework\PP1\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("+",BinOp("%",Int(10),Int(3)),BinOp("%",Int(5),Int(2))))
Result: 2
PS E:\Homework\PP1\Lab\Sample_Code>
```

Q.3.2. Scenario 2: include the \wedge operator for exponentiation.

Sample input: $2 \wedge 3$.

Result:




The screenshot shows the same IDE environment as the previous one, but with the `testcase.txt` tab containing the input `1 2 ^ 3`. The `TERMINAL` tab shows the following output:

```
PS E:\Homework\PP1\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("^",Int(2),Int(3)))
Result: 8
PS E:\Homework\PP1\Lab\Sample_Code>
```

Sample input: $2^3 + 5 * 2$.

Result:


```
tests >  testcase.txt
1 2 ^ 3 + 5 * 2

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL POSTMAN CONSOLE PORTS

PS E:\Homework\PP1\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("+",BinOp("^",Int(2),Int(3)),BinOp("*",Int(5),Int(2))))
Result: 18
❖ PS E:\Homework\PP1\Lab\Sample_Code> 
```

Sample input: $2^3 + 5^2$.

Result:

```
tests >  testcase.txt
1 2 ^ 3 + 5 ^ 2

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL POSTMAN CONSOLE PORTS

PS E:\Homework\PP1\Lab\Sample_Code> python run.py test
Complete jar file ANTLR : D:/antlr/antlr-4.13.2-complete.jar
Length of arguments : 1
-----
Running testcases...
Input accepted
-----
Run tests completely
This is ast string: Prog(BinOp("+",BinOp("^",Int(2),Int(3)),BinOp("^",Int(5),Int(2))))
Result: 33
```