Lab 6 - Abstract Syntax Tree - Report

Students:

1. Nguyen Huynh Thao My - ITCSIU21204
2. Pham Duc Dat - ITITIU20184

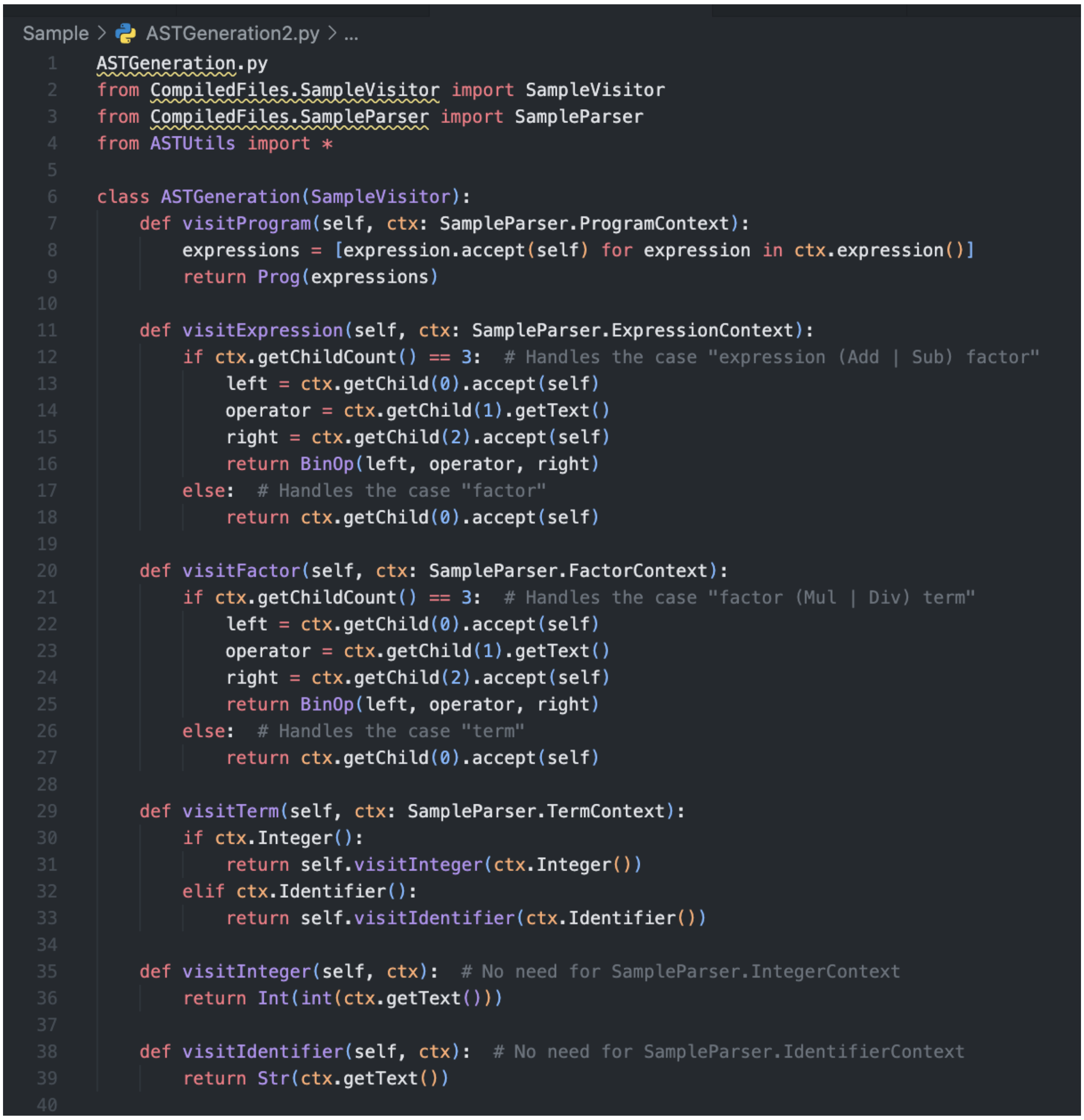
1 Overview

Lab 6 focuses on understanding and implementing Abstract Syntax Trees (AST) using ANTLR-generated parsers. The exercises explore the construction of ASTs from simple to complex grammar structures, covering basic arithmetic expressions, identifier handling, and nested operations. The lab demonstrates how to build and traverse ASTs using visitor patterns, showcasing the process of translating parsed input into a hierarchical tree structure. Through practical examples, this lab highlights the role of ASTs in representing the syntactic structure of source code, which is crucial for further tasks such as interpretation, compilation, or static analysis. By the end of this lab, students gain hands-on experience in developing and visualizing ASTs, reinforcing their understanding of how programming languages process and represent code internally.

2 Results

2.1 Exercise 3: Write code in "ASTGeneration.py" to parse AST trees with the following grammars.

```
program: (expression )*;
expression: expression (Add | Sub) factor | factor;
factor: factor (Mul | Div) term | term;
Add : '+';
Sub : '-';
Mul : '*';
Div : '/';
term: Integer | Identifier;
Integer: [0-9]+ ;
Identifier: [a-z]+ ;
```



Sample > ASTGeneration2.py > ...

```python
ASTGeneration.py
from CompiledFiles.SampleVisitor import SampleVisitor
from CompiledFiles.SampleParser import SampleParser
from ASTUtils import *

class ASTGeneration(SampleVisitor):
    def visitProgram(self, ctx: SampleParser.ProgramContext):
        expressions = [expression.accept(self) for expression in ctx.expression()]
        return Prog(expressions)

    def visitExpression(self, ctx: SampleParser.ExpressionContext):
        if ctx.getChildCount() == 3:  # Handles the case "expression (Add | Sub) factor"
            left = ctx.getChild(0).accept(self)
            operator = ctx.getChild(1).getText()
            right = ctx.getChild(2).accept(self)
            return BinOp(left, operator, right)
        else:  # Handles the case "factor"
            return ctx.getChild(0).accept(self)

    def visitFactor(self, ctx: SampleParser.FactorContext):
        if ctx.getChildCount() == 3:  # Handles the case "factor (Mul | Div) term"
            left = ctx.getChild(0).accept(self)
            operator = ctx.getChild(1).getText()
            right = ctx.getChild(2).accept(self)
            return BinOp(left, operator, right)
        else:  # Handles the case "term"
            return ctx.getChild(0).accept(self)

    def visitTerm(self, ctx: SampleParser.TermContext):
        if ctx.Integer():
            return self.visitInteger(ctx.Integer())
        elif ctx.Identifier():
            return self.visitIdentifier(ctx.Identifier())

    def visitInteger(self, ctx):  # No need for SampleParser.IntegerContext
        return Int(int(ctx.getText()))

    def visitIdentifier(self, ctx):  # No need for SampleParser.IdentifierContext
        return Str(ctx.getText())

```

Figure 1: Code for AST Generation Ex3

2.1.1 Code for Ex3

2.1.2 Explanation for Exercise 3: AST Generation for Complex Expressions

In Exercise 3, the AST is constructed for more complex expressions involving both arithmetic operations and identifiers. The following key node types are utilized:

- **Prog**: Represents the entire program as a list of expressions.
- **Int**: Represents an integer value.

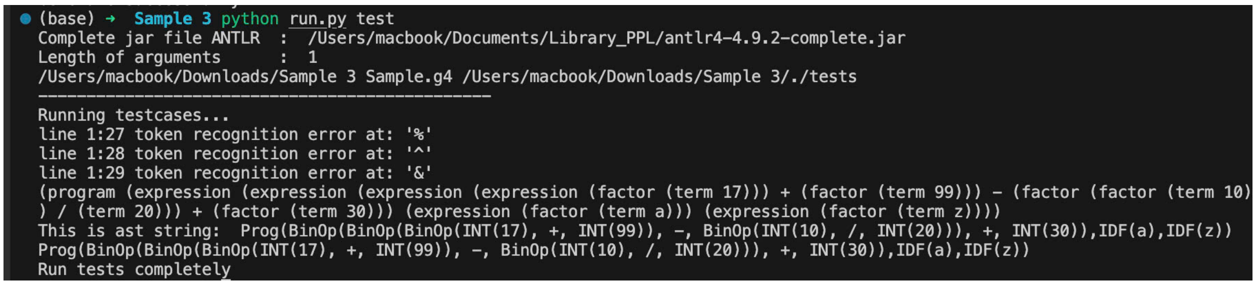
- **Str**: Represents an identifier or string.
- **BinOp**: Represents a binary operation (e.g., addition, subtraction, multiplication, division), connecting two expressions.

Process Overview:

- The `visitProgram` method processes all expressions in the program, encapsulating them in a `Prog` node.
- `visitExpression` handles binary operations by creating `BinOp` nodes for operations like addition or subtraction. If the expression consists of a single factor, it delegates to `visitFactor`.
- `visitFactor` processes multiplication and division operations, creating `BinOp` nodes similarly to `visitExpression`.
- `visitTerm` processes integer literals and identifiers, creating `Int` or `Str` nodes as appropriate.

Example Execution:

- For the input `"17 + 99 - 10 / 20 + 30"`, the parser generates an AST:

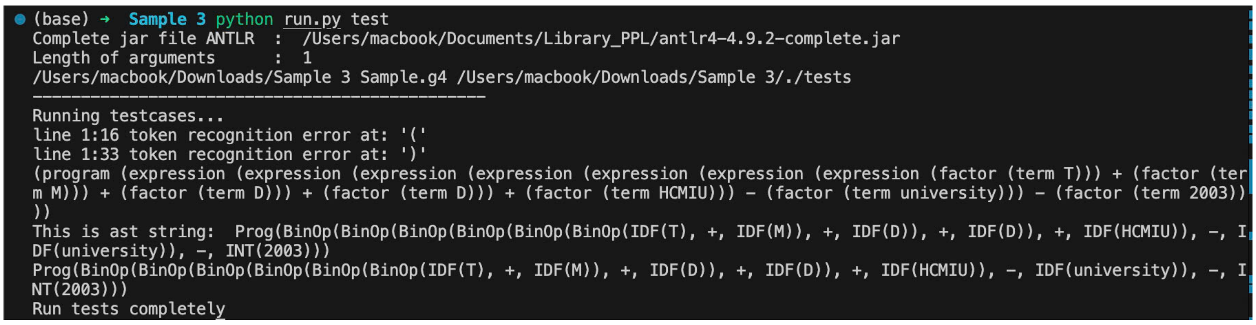


```
● (base) ➜  Sample 3 python run.py test
  Complete jar file ANTLR  :  /Users/macbook/Documents/Library_PPL/antlr4-4.9.2-complete.jar
  Length of arguments      :  1
  /Users/macbook/Downloads/Sample 3 Sample.g4 /Users/macbook/Downloads/Sample 3/./tests
  -----------------------------------------------
  Running testcases...
  line 1:27 token recognition error at: '%'
  line 1:28 token recognition error at: '^'
  line 1:29 token recognition error at: '&'
  (program (expression (expression (expression (expression (factor (term 17))) + (factor (term 99))) - (factor (factor (term 10)
  ) / (term 20))) + (factor (term 30))) (expression (factor (term a))) (expression (factor (term z))))
  This is ast string:  Prog(BinOp(BinOp(BinOp(INT(17), +, INT(99)), -, BinOp(INT(10), /, INT(20))), +, INT(30)),IDF(a),IDF(z))
  Prog(BinOp(BinOp(BinOp(INT(17), +, INT(99)), -, BinOp(INT(10), /, INT(20))), +, INT(30)),IDF(a),IDF(z))
  Run tests completely
```

Figure 2: Output for Example 01

- For the input `"T + M + D + D + (HCMIU-university) - 2003"`, the AST generated is:

These ASTs represent the hierarchical structure of the expressions, with each operation forming a `BinOp` node that connects its operands. Identifiers and integers are captured as `Str` and `Int` nodes, respectively, demonstrating the parser's ability to handle



```
● (base) ➜  Sample 3 python run.py test
  Complete jar file ANTLR  :  /Users/macbook/Documents/Library_PPL/antlr4-4.9.2-complete.jar
  Length of arguments      :  1
  /Users/macbook/Downloads/Sample 3 Sample.g4 /Users/macbook/Downloads/Sample 3/./tests
  ----------------------------------------------
  Running testcases...
  line 1:16 token recognition error at: '('
  line 1:33 token recognition error at: ')'
  (program (expression (expression (expression (expression (expression (expression (expression (factor (term T))) + (factor (ter
  m M))) + (factor (term D))) + (factor (term D))) + (factor (term HCMIU))) - (factor (term university))) - (factor (term 2003))
  ))
  This is ast string:  Prog(BinOp(BinOp(BinOp(BinOp(BinOp(BinOp(IDF(T), +, IDF(M)), +, IDF(D)), +, IDF(D)), +, IDF(HCMIU)), -, I
  DF(university)), -, INT(2003)))
  Prog(BinOp(BinOp(BinOp(BinOp(BinOp(BinOp(IDF(T), +, IDF(M)), +, IDF(D)), +, IDF(D)), +, IDF(HCMIU)), -, IDF(university)), -, I
  NT(2003)))
  Run tests completely
```

Figure 3: Output for Example 02

both numeric and textual data within expressions.