Project 1 Report
Name: Fanyue Zhu Student id: 522031910547

1 Objectives

In the second part, we need to build a grammar parser that constructs the obtained lexical unit sequences into an Abstract Syntax Analysis Tree (AST). This includes parsing function declarations and calls, Tensor variable declarations, and Tensor binary arithmetic expressions, and outputting error messages for illegal formats.

2 Implementations

2.1 Grammar analysis function implementation

2.1.1 parseDeclaration()

1. Keyword 'var' recognition:

```
        if (lexer.getCurToken() != tok_var)
          return parseError<VarDeclExprAST>("var", "to begin variable
    declaration");
        lexer.consume(tok_var);

```

2. Three methods of initialization support: First check if the next token is '¡' to determine whether it's the third method. Then consume the identifier. Check '¡' again to distinguish the second way.

```
        std::unique_ptr<VarType> type;
        if (lexer.getCurToken() == '<') {
          type = parseType();
          if (!type)
            return nullptr;
        } //check the third one

        if (lexer.getCurToken() != tok_identifier)
          return parseError<VarDeclExprAST>("identifier", "in variable
    declaration");
        id = std::string(lexer.getId());
        lexer.consume(tok_identifier);
        //lexer.getNextToken();

      // Type is optional, it can be inferred
        if (lexer.getCurToken() == '<') {
          type = parseType();
          if (!type)
            return nullptr;
        } // check the second one

        if (!type)
          type = std::make_unique<VarType>();
        lexer.consume(Token('='));
        auto expr = parseExpression();
        return std::make_unique<VarDeclExprAST>(std::move(loc), std::move(id),
                                                        std::move(*type), std::move(
    expr));

```

2.1.2 parseIdentifierExpr()

1. Get and eat identifier:

```
        auto loc=lexer.getLastLocation();
        std::string id=std::string(lexer.getId());
        lexer.consume(tok_identifier);

```

2. Determine how the call is made: Determine whether it is an identifier, a normal function call or a call to the built-in function print. If it is only a variable name, return its corresponding AST. In the case of a function call, parse the arguments one by one with **parseExpression()**. In the case of**print()**, make sure that there is only one parameter inside it. Return their corresponding AST.

```
        if(lexer.getCurToken()!='(')
        {
          return std::make_unique<VariableExprAST>(std::move(loc),id);
        }

        lexer.consume(Token('('));

        std::vector<std::unique_ptr<ExprAST>> args;
        if(lexer.getCurToken()!=')')
        {
          while(true)
          {
            auto arg=parseExpression();
            if(!arg)
              return nullptr;
            args.push_back(std::move(arg));
            if(lexer.getCurToken()==')')
              break;
            if(lexer.getCurToken()!=',')
              return parseError<ExprAST>(")", "to close function call");
            lexer.consume(Token(','));
          }
        }
        lexer.consume(Token(')'));
        if(id=="print")
        {
          if(args.size()!=1)
            return parseError<ExprAST>("only one argument", "in print function
    ");
          return std::make_unique<PrintExprAST>(std::move(loc),std::move(args
    [0]));
        }
        return std::make_unique<CallExprAST>(std::move(loc),id,std::move(args)
    );

```

2.1.3 parseBinOpRHS()

Recursively parse the right hand side of a binary expression, until it's all merged by lhs.

```
    auto loc=lexer.getLastLocation();

    while (true)
    {
      int tokPrec=getTokPrecedence();
      if(tokPrec<exprPrec)
        return lhs;

      int binOp=lexer.getCurToken();
      lexer.getNextToken();

      auto rhs=parsePrimary();
      if(!rhs)
        return nullptr;

      int nextPrec=getTokPrecedence();
      if(tokPrec<nextPrec)
      {
        rhs=parseBinOpRHS(tokPrec+1,std::move(rhs));
        if(!rhs)
          return nullptr;
      }

      lhs=std::make_unique<BinaryExprAST>(std::move(loc),binOp,std::move(lhs),std::
   move(rhs));
    }
```

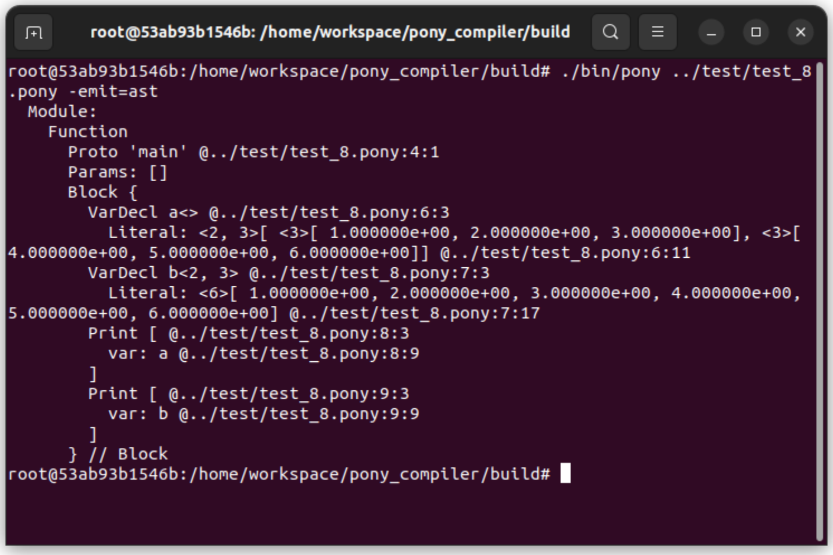
2.1.4 getTokPrecedence()

Add a judgment on @ in case to set the priority the same as *.

```
    switch (static_cast<char>(lexer.getCurToken())) {
    case '-':
      return 20;
    case '+':
      return 20;
    case '*':
      return 40;
    case '@':
      return 40;
    default:
      return -1;
    }
```

3 Results

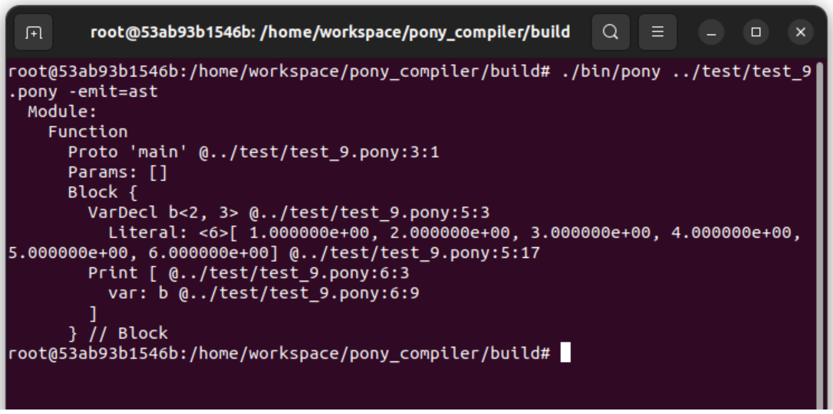
test8.pony



```
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_8
.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_8.pony:4:1
      Params: []
      Block {
        VarDecl a<> @../test/test_8.pony:6:3
          Literal: <2, 3>[ <3>[ 1.000000e+00, 2.000000e+00, 3.000000e+00], <3>[
4.000000e+00, 5.000000e+00, 6.000000e+00]] @../test/test_8.pony:6:11
        VarDecl b<2, 3> @../test/test_8.pony:7:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00] @../test/test_8.pony:7:17
        Print [ @../test/test_8.pony:8:3
          var: a @../test/test_8.pony:8:9
        ]
        Print [ @../test/test_8.pony:9:3
          var: b @../test/test_8.pony:9:9
        ]
      } // Block
root@53ab93b1546b:/home/workspace/pony_compiler/build# 
```

Figure 1: test8

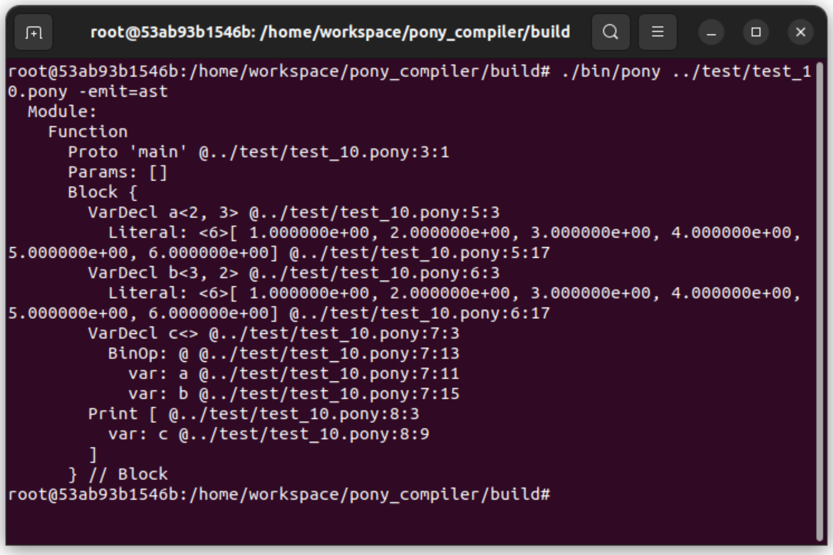
test9.pony: The third method of initialization



```
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_9
.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_9.pony:3:1
      Params: []
      Block {
        VarDecl b<2, 3> @../test/test_9.pony:5:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00] @../test/test_9.pony:5:17
        Print [ @../test/test_9.pony:6:3
          var: b @../test/test_9.pony:6:9
        ]
      } // Block
root@53ab93b1546b:/home/workspace/pony_compiler/build# 
```

Figure 2: test9

test10.pony: New matrix multiplication operations @



```
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_1
0.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_10.pony:3:1
      Params: []
      Block {
        VarDecl a<2, 3> @../test/test_10.pony:5:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00] @../test/test_10.pony:5:17
        VarDecl b<3, 2> @../test/test_10.pony:6:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00] @../test/test_10.pony:6:17
        VarDecl c<> @../test/test_10.pony:7:3
          BinOp: @ @../test/test_10.pony:7:13
            var: a @../test/test_10.pony:7:11
            var: b @../test/test_10.pony:7:15
        Print [ @../test/test_10.pony:8:3
          var: c @../test/test_10.pony:8:9
        ]
      } // Block
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 3: test10

4 Drawback

I don't know why that in the Var Declaration part my code will output the identifier along with the tensor shape. Sorry I can't solve the problem so I submit with it unsolved.