Project 1 Report

Name: Fanyue Zhu Student id: 522031910547
Email:lokawa0.0@sjtu.edu.cn

Contents

Part 1

1.1 Objectives

Build a lexical analyser to identify the various lexical units (tokens) in the Pony language, including keywords (such as var, def and return), special symbols, numbers and variable/function names. The Token is obtained through a function that determines its legitimacy and outputs an error message for illegal formats.

1.2 Implementation

1.2.1 Lexical Analysis Function Implementation

1.2.1.1 getNextChar()

First determine if it is an end of line. If not, directly return to the next character; otherwise, read the next line and update the rows and columns, if the next line is empty then the proof is EOF, otherwise return to the normal first character.

```
int getNextChar()
{
    if (curCol == curLineBuffer.size()) {
      curLineBuffer = readNextLine();
      curCol = 0;
      curLineNum++;
      if (curLineBuffer.empty()) {
        return EOF;
      }
    }
    return curLineBuffer[curCol++];
}
```

1.2.1.2 getTok()

1. Keyword Identification: As part of identifier recognition, determines if it is the same as the keyword after recognition.

```
    if (strcmp(idStr.c_str(), "return") == 0)
    {
      return tok_return;
    }
    if (strcmp(idStr.c_str(), "var") == 0)
    {
      return tok_var;
    }
    if (strcmp(idStr.c_str(), "def") == 0)
    {
      return tok_def;
```

```
    }

```

2. Identifier Identification:

   (a) Determines if lastChar is a _ or a letter, if it is it goes to identifier judgement. Define string idStr, loop to collect token characters and judge whether it is legal and whether it is a keyword, if it meets the condition then assign the string idStr to identifieraStr to store for subsequent output.

```
    if (isalpha(lastChar) || lastChar == '_')
    {
      std::string idStr;
      do {
        idStr += lastChar;
        lastChar = Token(getNextChar());
      } while (isalnum(lastChar) || lastChar == '_');

      //Legitimacy checking and keyword detection

      identifierStr = idStr;
      return tok_identifier;
    }

```

   (b) Illegal identifier recognition
   Use **find()** to check whether there are continuous '_'

```
      // no continuous '_'
      if (idStr.find("__") != std::string::npos)
      {
        llvm::errs() << "Identifier <" << idStr << "> contains continuous
    '_'\n";
        identifierStr ="ERROR_IDENTIFIER";
        return tok_identifier;
      }

```

   Use **find()** to check whether there's '_' at the beginning.

```
      // no '_' at the beginning
      if (idStr.find("_") == 0)
      {
        llvm::errs() << "Identifier <" << idStr << "> start with '_'\n";
        identifierStr = "ERROR_IDENTIFIER";
        return tok_identifier;
      }

```

   Use **find_first_of()** and **find_first_not_of()** to check whether there's number exits but not only at the end.

```
      // digit only at the end
      if (idStr.find_first_of("0123456789") != std::string::npos)
      {
        if (idStr.find_first_not_of("0123456789", idStr.find_first_of
    ("0123456789")) != std::string::npos) {
          llvm::errs() << "Identifier <" << idStr << "> end with number\n
    ";
```

```
          identifierStr = "ERROR_IDENTIFIER";
          return tok_identifier;
        }
      }

```

3. Illegal Number Identification: Similar to the illegal identifier detection above

```
      // no continuous '.'
      if (numStr.find("..") != std::string::npos)
      {
        llvm::errs() << "Invalid number: " << numStr << "\n";
        identifierStr = "ERROR_NUMBER";
        return tok_identifier;
      }
      // no more than one '.'
      if (numStr.find_first_of(".") != numStr.find_last_of("."))
      {
        llvm::errs() << "Invalid number: " << numStr << "\n";
        identifierStr = "ERROR_NUMBER";
        return tok_identifier;
      }
      // no '.' at the beginning
      if (numStr.find(".") == 0)
      {
        llvm::errs() << "Invalid number: " << numStr << "\n";
        identifierStr = "ERROR_NUMBER";
        return tok_identifier;
      }

```

1.2.2 Lexical Analysis Verification Procedure Implementation

Define a vector of type string and traverse the input file with Lexer to get the corresponding tok return value. If it is a keyword, character or EOF, tokenName is the corresponding value; if it is an identifier, use **getId()** to get the name of the identifier; if it is a number, use **getValue()** to get the corresponding value, and convert the floating-point number to a string type to remove the trailing redundant 0.

It is worth noting that the lexer pointer is already pointing to the first token before traversal, and it is necessary for the token to first use **getCurToken()** to get the token it is currently pointing to.

```
std::vector<std::string> tokens;
auto token = lexer.getCurToken();
while (true)
{
  std::string tokenName;
  switch (token)
  {
  case Token::tok_eof:
    tokenName = "EOF";
    break;
  case Token::tok_return:
    tokenName = "return";
    break;
  case Token::tok_def:
```

```
      tokenName = "def";
      break;
    case Token::tok_var:
      tokenName = "var";
      break;
    case Token::tok_identifier:
      tokenName = std::string(lexer.getId());
      break;
    case Token::tok_number:
      tokenName = std::to_string(lexer.getValue());
      tokenName = tokenName.substr(0, tokenName.find_last_not_of('0') + 1);
      if (tokenName.back() == '.')
      {
        tokenName=tokenName.substr(0, tokenName.size()-1);
      }
      break;
    default:
      tokenName = std::string(1, token);
      break;
    }
    tokens.push_back(tokenName);
    //break when eof
    if (token == Token::tok_eof)
    {
      break;
    }
    token=lexer.getNextToken();
  }
  //print the token by order
  for (auto &token : tokens) {
    llvm::outs() << token << " ";
  }
  llvm::outs() << "\n";
```

Part 2

2.1 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.

2.2 Implementations

2.2.1 Grammar analysis function implementation

2.2.1.1 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));

```

2.2.1.2 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));

```

2.2.1.3 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));
  }
```

2.2.1.4 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;
  }
```

Part3

3.1 Objectives

The Pony language has a built-in transpose function that performs a transpose operation on a matrix. However, transposing the same matrix twice results in the original matrix, which is not transposed. The transpose operation on the matrix is implemented using nested for loops, which is an important factor in the speed of a program. Therefore, it is necessary to detect and eliminate this redundant code.
Also, in MLIR, high-level languages are converted from high to low into intermediate representations (called dialect) at different levels of abstraction, generating the corresponding intermediate code, and eventually generating the bottom-level executable code. intermediate code, and finally the lowest level executable code. In order to execute an application in the Pony language, we need to do the following

1. parse the Pony program (.pony) file and generate the corresponding pony dialect representation
2. convert the pony dialect into some of MLIR's built-in dialects (arith, memref, and affine)
3. convert affine dialect to executable llvm dialect

This experiment only requires students to implement the conversion of pony.gemm to MLIR's built-in dialects.

3.2 Implementations

3.2.1 Code Optimisation - Redundant Transpose Elimination

3.2.1.1 struct `SimplifyRedundantTranspose`

Use `getOperand()` to get the input of current transpose, then check whether the input is defined by another transpose by `getDefiningOp()`. If so, remove the redundant transpose with `rewriter.replaceOp()`.

```
    Value input = op.getOperand();//get input

    TransposeOp transposeOp = input.getDefiningOp<TransposeOp>();
    if (!transposeOp) return failure();//check redefination

    rewriter.replaceOp(op, transposeOp.getOperand());
    return success();//remove redundant transpose
```

3.2.2 Intermediate Code Generation

3.2.2.1 `GemmOp::inferShapes()`

```
    auto lhsTy = getOperand(0).getType().cast<RankedTensorType>();
    auto rhsTy = getOperand(1).getType().cast<RankedTensorType>();
    auto lhsShape = lhsTy.getShape();
    auto rhsShape = rhsTy.getShape();

    getResult().setType(RankedTensorType::get({lhsShape[0], rhsShape[1]},
                                              lhsTy.getElementType()));
```

3.2.2.2 struct `GemmOpLowering`

According to the shape, we can update upper bounds. Information about the shape of the matrix can be obtained from the type of the operand.

```
    auto ashape = operands[0].getType().cast<MemRefType>().getShape();//M*K
    auto bshape = operands[1].getType().cast<MemRefType>().getShape();//K*N

    upperBounds[0] = ashape[0];//M
    upperBounds[1] = bshape[1];//N
    upperBounds[2] = ashape[1];//K
```

Then build the affine loop. Load elements from A and B matrices and the current value from C, then multiply and accumulate and store the result back to C.

```
    buildAffineLoopNest(
        rewriter, loc, lowerBounds, upperBounds, steps,
        [&](OpBuilder &nestedBuilder, Location loc, ValueRange ivs) {
          typename pony::GemmOp::Adaptor gemmAdaptor(operands);


          Value i=ivs[0];//for M
          Value j=ivs[1];//for N
          Value k=ivs[2];//for K

          auto loadedA=nestedBuilder.create<AffineLoadOp>(loc,gemmAdaptor.getLhs(),ValueRange{i,k});
          auto loadedB=nestedBuilder.create<AffineLoadOp>(loc,gemmAdaptor.getRhs(),ValueRange{k,j});
          auto loadedC=nestedBuilder.create<AffineLoadOp>(loc,alloc,ValueRange{i,j});//load value

          auto mul=nestedBuilder.create<arith::MulFOp>(loc,loadedA,loadedB);
          auto add=nestedBuilder.create<arith::AddFOp>(loc,loadedC,mul);//multiply and accumulate

          nestedBuilder.create<AffineStoreOp>(loc,add,alloc,ValueRange{i,j});//store back

        });
```

Results and Achievements

4.1 Results

All test case results are listed as below.

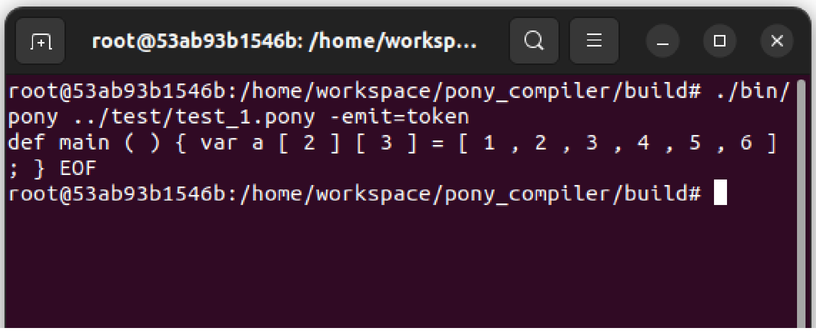

Figure 1: Result of test1

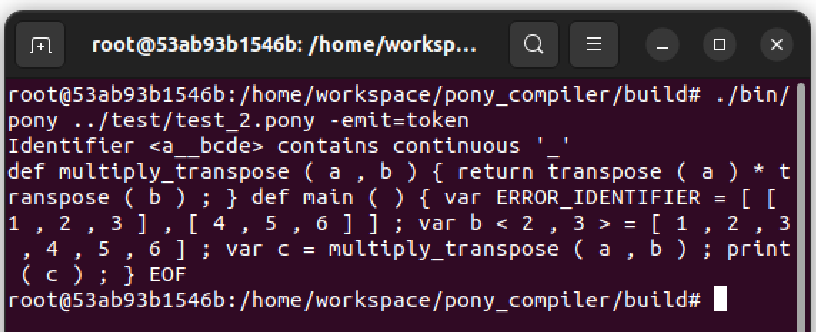

Figure 2: Result of test2

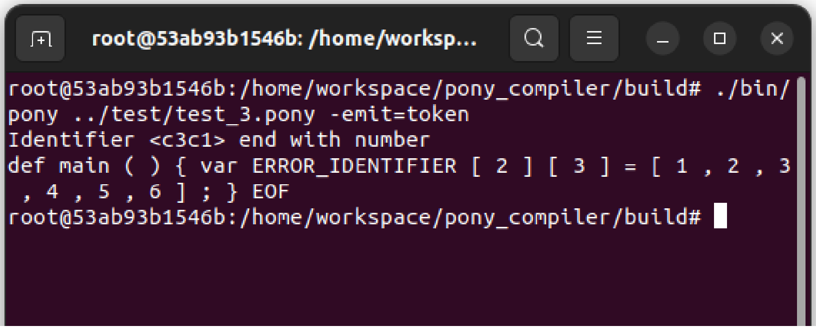

Figure 3: Result of test3

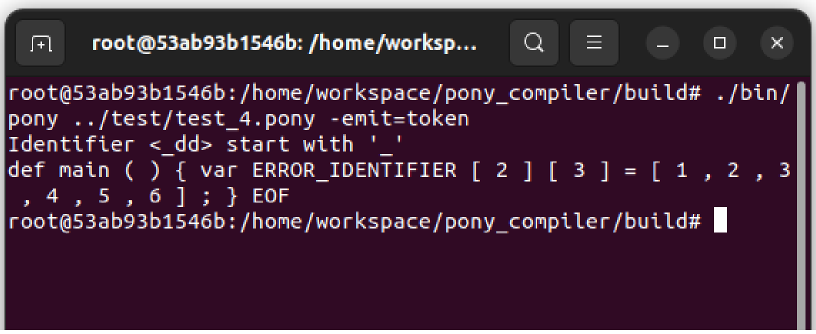

Figure 4: Result of test4

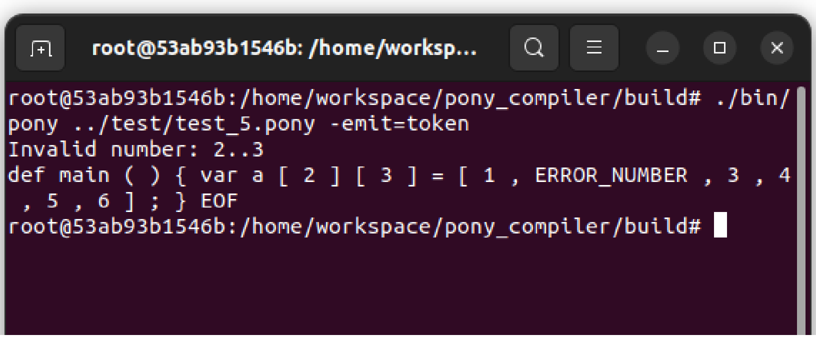

Figure 5: Result of test5

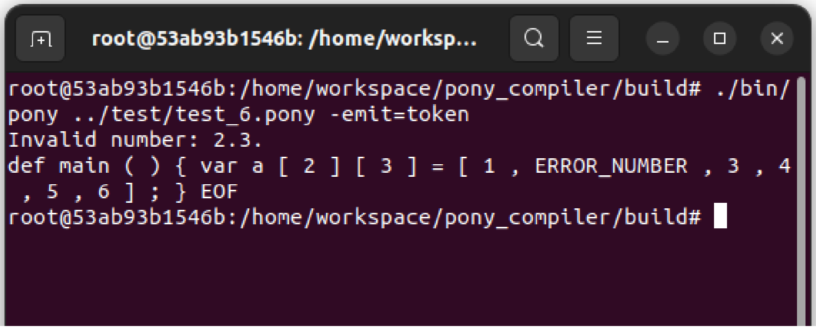

Figure 6: Result of test6

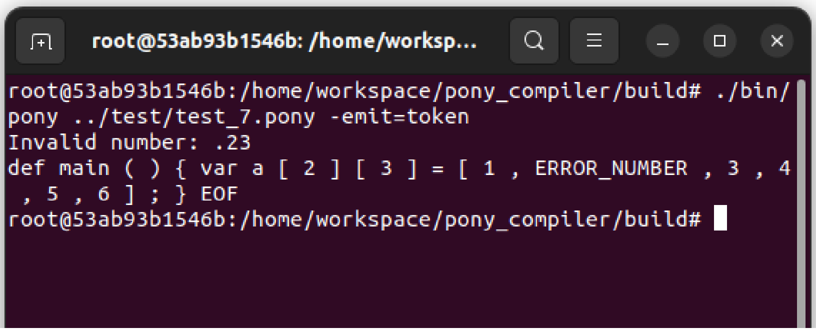

Figure 7: Result of test7

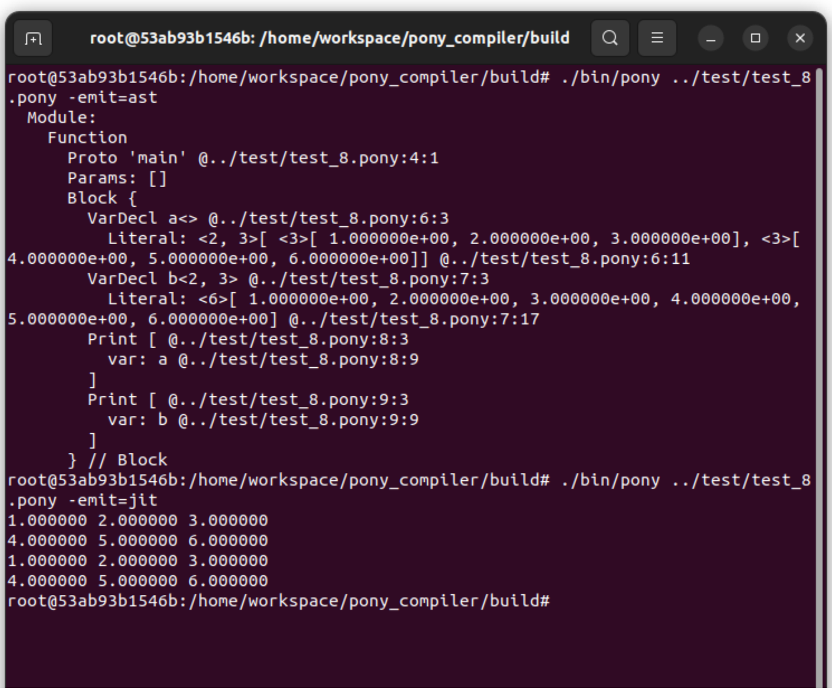

Figure 8: Result of test8

Figure 9: Result of test9

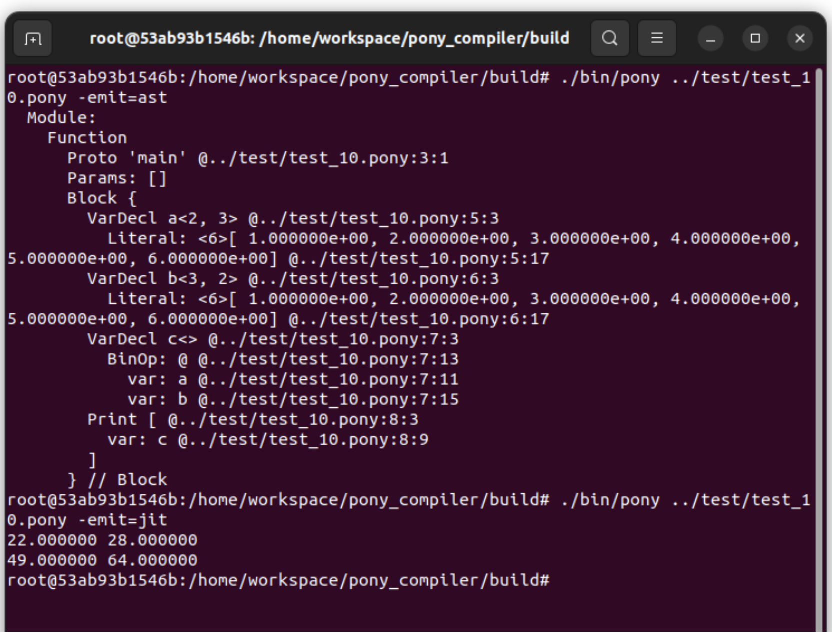

Figure 10: Result of test10

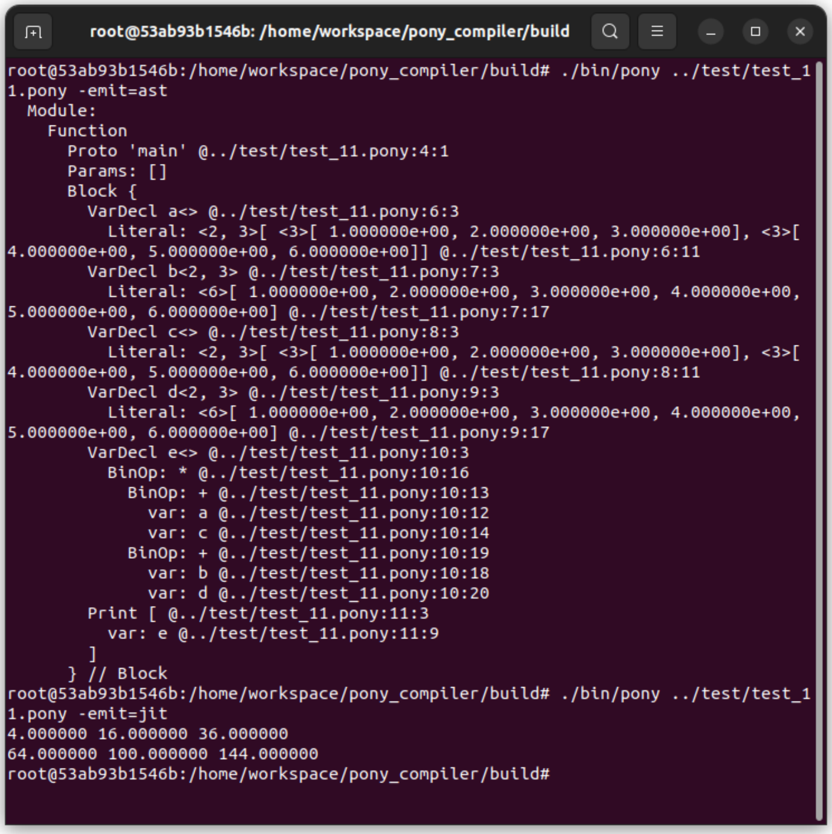

Figure 11: Result of test11

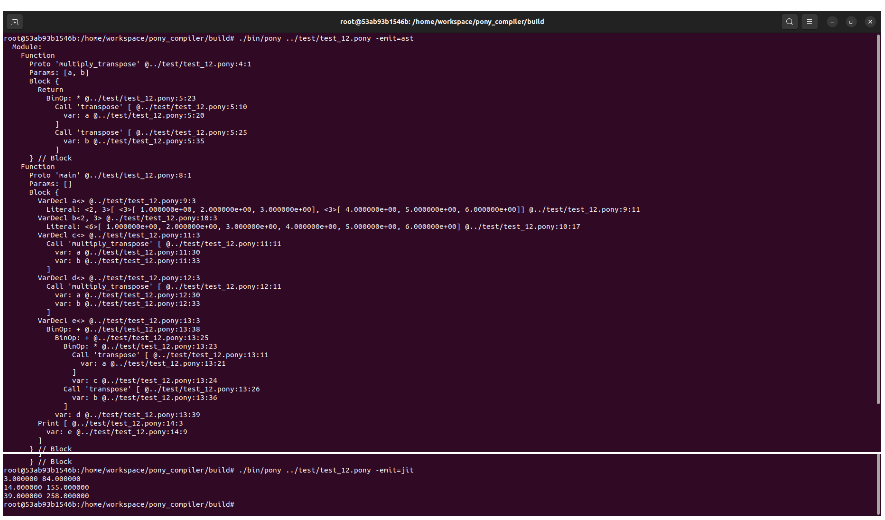

Figure 12: Result of test12

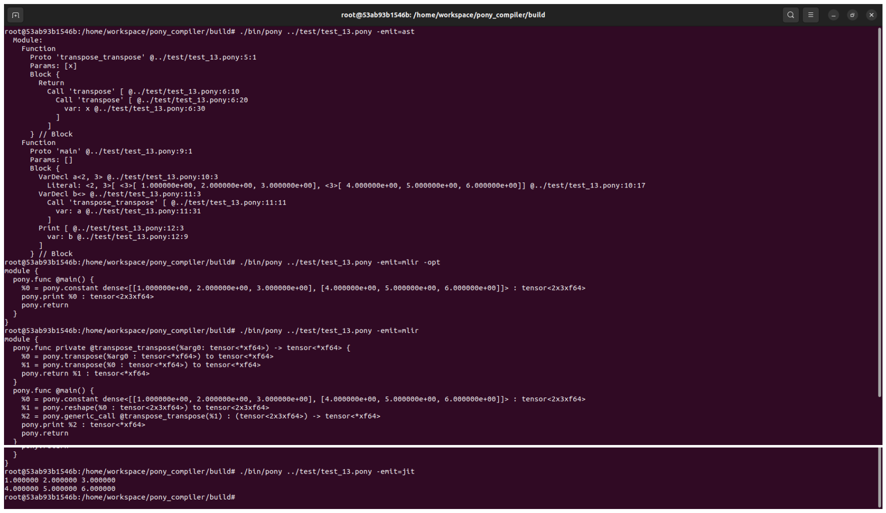

Figure 13: Result of test13

4.2 Achievements

After three parts of experiments, we've finished the compiler of language pony, simulated the compilation process and gained a deeper insight into how compiler works.
Actually, it's a little difficult for me to accomplish the work. Most of the work needs imitation of other functions rather than the theory I learned in class, but it's hard for one without any previous knowledge of llvm, mlir and so on libraries and their corresponding wrapper functions to understand the code given. Like the affine loop in Part3, I spent plenty time on figuring out the defination of the variables. I think it's a point that could be perfected.
All in all, this project is very inspiring and I 've learned a lot.

Acknowledgement

Thank our teachers and teaching assistants for providing support and answering questions to enable us to complete the project from scratch.