MLIR 编译框架的使用与探索
第二部分：语法分析

Junhao Dai daijunhao@sjtu.edu.cn

May 18, 2024

1 简介

1.1 实验目的

本次实验旨在探索并使用编译基础设施框架 MLIR(Multi-Level Intermediate Representation)，利用课堂所学习的词法分析、语法分析以及代码优化与生成的知识，去完善基于张量的 Pony 语言，从编译的角度去解析 Pony 语言。

在第二部分，我们需要构建一个语法分析器，将获得的词法单元序列构建成一棵抽象语法分析树 (AST)。具体包括解析函数的声明和调用，Tensor 变量的声明以及 Tensor 的二元运算表达式等，并针对非法格式输出错误信息。

1.2 实验环境

- 虚拟机软件：VMware Workstation Pro 16.2.3
- 操作系统：Ubuntu 22.04 LTS
- 依赖软件：git version 2.34.1，cmake version 3.22.1，gcc version 11.4.0

2 功能实现

2.1 成员函数 parseDeclaration()

该函数实现对语法变量“var”的识别，以及对变量名“identifier”以及其对应的初始化形式的 tensor shape 的识别。语法分析器需要支持以下三种初始化形式：

- var a = [[1, 2, 3], [4, 5, 6]];
- var a<2,3> = [1, 2, 3, 4, 5, 6];
- var<2,3> a = [1, 2, 3, 4, 5, 6];

核心代码如下：

```
auto loc = lexer.getLastLocation();
std::string id;
if (lexer.getCurToken() != tok_var)
    return parseError<VarDeclExprAST>("var", "in variable declaration");
```

```
lexer.getNextToken(); // eat var

std::unique_ptr<VarType> type; // Type is optional, it can be inferred
if (lexer.getCurToken() != tok_identifier && lexer.getCurToken() != '<')
    return parseError<VarDeclExprAST>("identifier or type", "in variable and type declaration");
else if (lexer.getCurToken() == tok_identifier) {
    id = lexer.getId().str();
    lexer.getNextToken(); // eat identifier

    if (lexer.getCurToken() == '<') {
        type = parseType();
        if (!type)
            return nullptr;
    }
} else if (lexer.getCurToken() == '<') {
    type = parseType();
    if (!type)
        return nullptr;
    // lexer.getNextToken(); // eatype

    if (lexer.getCurToken() != tok_identifier)
        return parseError<VarDeclExprAST>("identifier", "in variable declaration");
    id = lexer.getId().str();
    lexer.getNextToken(); // eat identifier
}

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 成员函数 parseIdentifierExpr()

此函数中，我们需要实现一个解析标识符语句的功能，其可以是简单的变量名，也可以用于函数调用，形式为 ::=identifier 或者 ::=identifier'('expression')' 。在我的代码中，首先获取语句开头的 identifier，然后判断下一个 token 是否为'('，若是则将其视为是一个函数调用，并读取后面的参数列表；若不是则直接返回 Variable 的 AST。读取参数时将所有参数存入 args 数组中，最后需要先判断函数是否为打印输出函数 print ，若是则判断参数是否只有一个，若为多个或无参数返回响应报错，最后返回函数调用表达式的 AST。核心代码如下：

```
std::string identifier = lexer.getId().str();
auto loc = lexer.getLastLocation();

if (lexer.getNextToken() != '(')
```

```
    return std::make_unique<VariableExprAST>(std::move(loc), identifier);

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>("') or ,", "to close function call or input arguments");
        lexer.getNextToken(); // eat ,
    }
}
lexer.consume(Token(')'));

if (identifier == "print") {
    if (args.size() != 1)
        return parseError<ExprAST>("one argument", "for print statement");
    return std::make_unique<PrintExprAST>(std::move(loc), std::move(args[0]));
}

return std::make_unique<CallExprAST>(std::move(loc), identifier, std::move(args));
```

2.3 成员函数 parseBinOpRHS(int exprPrec, std::unique_ptr<ExprAST> lhs)

在此函数中我们需要使用递归的方法来解析一个二元表达式。表达式可能像 '+'primary '−'primary 或者 ('+'primary)∗ 这样的表达式。第一个参数 (exprPrec) 表示当前二元运算符的优先级，第二个参数 (lhs) 表示表达式的左侧表达式。核心代码如下：

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

    int binOp = lexer.getCurToken();
    lexer.consume(Token(binOp));
    auto loc = lexer.getLastLocation();

    auto rhs = parsePrimary();
```

```
    if (!rhs)
        return parseError<ExprAST>("primary", "in binary expression");

    int nextPrec = getTokPrecedence();
    if (nextPrec < 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));
}
```

此外，我们还需要增加对矩阵乘法 @ 的支持，其优先级和矩阵点乘 * 相同，修改 getTokPrecedence() 部分代码如下：

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

3 实验结果

在对语法分析器构建完毕后，可以通过运行测试用例 test_8 至 test_10 来检查词法分析器的正确性。测试结果如图1所示。从输出结果不难发现，语法分析器正确地得到了输入语句的解析，test 中正确解析了 var<2, 3> a = [1, 2, 3, 4, 5, 6] 的变量声明形式，test_10 中正确解析了矩阵的 @ 运算。

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../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@f965a549d8ea:/home/workspace/pony_compiler/build# ../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@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_10.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:7:15
            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
```

test_8 到 test_10 测试结果

4 总结

总的来说本次实验过程比较顺利，得益于老师和助教在微信群中的悉心答疑以及精心准备的实验指导书。最开始我在忽略了 docker 容器的重新启动与连接，导致编译失败，这也让我对 docker 的使用更加熟练。之后的编译错误由于输出报错过长也让我无从下手。我采用将报错输出到文件的方式来逐步解决报错的问题，最终编译成功。但在对函数声明和变量声明部分始终报错，因此我不断排查代码问题，最终我发现由于识别的 identifier 没有获取下一个 token 而导致了此次错误。经过这次实验我对语法分析过程以及 docker 的使用都有了更好的理解。