语法分析实验报告

胡基宸 521021910143

1. 思路和方法

① 解析变量的声明，实现成员函数 parseDeclaration()并扩展成员函数 parseType()，要求:

1). 语法变量必须以“var”开头，后面为变量名及 tensor shape

2). 语法分析器已支持以下两种初始化形式，以一个二维矩阵为例:

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

需要同学们额外支持第三种新的形式:

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

parseVarDeclaration 函数用于对变量声明的表达式进行语法分析
变量声明的标准形式为： var identifier type = expression
Type 表示的是矩阵的维数，可以忽略
首先检查当前 Token 是否为 var,如果不是就在控制台报错, 如果是就 eat 掉当前的 Token, 也就是调用 lexer.getNextToken()。
然后检查当前 Token 是否为 identifier, 如果不是就在控制台报错, 如果是就用 lexer.getId() 处理。
接着检查是否有 type，只需要判断 < 和 [就行了。

```
// TODO: check to see if this is a 'var' declaration
//       If not, report the error with 'parseError', otherwise eat 'var'
/*
 *S
 *  Write your code here.
 *
 */
if (lexer.getCurToken() != tok_var)
  return parseError<VarDeclExprAST>("var", "in variable declaration");
lexer.getNextToken(); // eat var
```

```
// TODO: check to see if this is a variable name(identifier)
//       If not, report the error with 'parseError', otherwise eat the variable name
/*
 *
 *  Write your code here.
 *
 */
if (lexer.getCurToken() != tok_identifier)
    return parseError<VarDeclExprAST>("identified", "after 'var' declaration");
// eat the variable name
std::string id(lexer.getId());
lexer.getNextToken(); // eat id

std::unique_ptr<VarType> type; // Type is optional, it can be inferred
```

```
  // TODO: modify the code to additionally support the third method: var a[][] = ...
  if (lexer.getCurToken() == '<' || lexer.getCurToken() == '[') {
    type = parseType();
    if (!type)
      return nullptr;
  }

  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));
}
```

parseType 函数用于对变量维数进行语法分析。开始已经有了<>形式，我们需要加上[ ]的形式，也就是在[ ]的判断中，循环处理[number]的形式。

```
    // recursively process []
    auto type = std::make_unique<VarType>();
    while (lexer.getCurToken() == '[') {
      lexer.getNextToken(); // eat [
      if (lexer.getCurToken() == tok_number) {
        type->shape.push_back(lexer.getValue());
        lexer.getNextToken();
      }
      if (lexer.getCurToken() != ']')
        return parseError<VarType>("]", "to end shape");
      lexer.getNextToken(); // eat ]
    }
    return type;
  }
}
```

② 解析函数内的部分常用表达式，具体要求为：

1). 解析标识符语句，其可以是简单的变量名，也可以用于函数调用。要求实现成员函数 parseIdentifierExpr()

2). 解析矩阵的二元运算表达式，需要考虑算术符号的优先级。要求实现成员函数 parseBinOpRHS()

解析标识符语句，其可以是简单的变量名，也可以用于函数调用。具有以下形式：

::= identifier

::= identifier '(' expression ')'

在 parseCallExpr 函数中解析（）中的各个 arguments，对于错误的形式，比如缺少必须的‘，’和‘）’，以及对于 print 函数而言，argument 只能有一个，要特殊处理。

```
std::unique_ptr<ExprAST> parseCallExpr(llvm::StringRef name,
                                       const Location &loc) {
  lexer.consume(Token('('));
  std::vector<std::unique_ptr<ExprAST>> args;//the arguments
  if (lexer.getCurToken() != ')') {
    while (true) {//recursively get the args
      if (auto arg = parseExpression())
        args.push_back(std::move(arg));
      else
        return nullptr;

      if (lexer.getCurToken() == ')')
        break;

      if (lexer.getCurToken() != ',')
        return parseError<ExprAST>(", or )", "in argument list");
      lexer.getNextToken();
    }
  }
  lexer.consume(Token(')'));

  // for the print, there cant only be one argument
  if (name == "print") {
    if (args.size() != 1)
      return parseError<ExprAST>("<single arg>", "as argument to print()");

    return std::make_unique<PrintExprAST>(loc, std::move(args[0]));
  }

  return std::make_unique<CallExprAST>(loc, std::string(name),
                                       std::move(args));
}
```

```
std::unique_ptr<ExprAST> parseIdentifierExpr() {
  /*
   *
   *  Write your code here.
   *
   */
  std::string name(lexer.getId());

  auto loc = lexer.getLastLocation();
  lexer.getNextToken(); // eat identifier.

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

  // function call.
  return parseCallExpr(name, loc);
}
```

在 parseBinOpRHS()解析矩阵的二元运算表达式

```cpp
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>("expression", "to complete binary operator");

  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. 结果验证

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_12 来检查语法分析器的正确性。

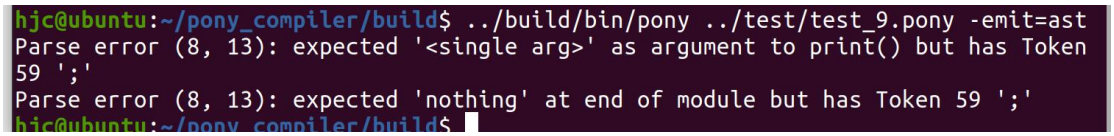
以 test_8 为例，验证语法分析器功能的正确性，输出 AST（-emit=ast）：

```
hjc@ubuntu:~/pony_compiler/build$ cmake --build . --target pony
[9/9] Linking CXX executable bin/pony
hjc@ubuntu:~/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
        ]
      } // Block
```

执行以下指令查看输入程序的运行结果（-emit=jit）：

```
hjc@ubuntu:~/pony_compiler/build$ ../build/bin/pony ../test/test_8.pony -emit=jit
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
```

以 test_9 为例，验证语法分析器能否正确解析非法的 print 形式：



可以看到，对于这几个测试用例，我们能够正确地输出语法分析得到的 AST，也可以输出程序的运行结果。

3. 遇到的问题与解决方法

1. 有一个地方忘记 eat 掉括号，导致一直报错。
2. 循环获取函数参数是开始少加了一个 break 导致死循环。