编译原理课程大作业

第二部分：语法分析

常烁晨
2023.5.4

一、摘要

本实验在第一部分词法分析的基础上，实现语法分析功能。在第一部分，我们主要修改了 lexer 的代码，构建了一个能完成解析和存储输入流中 Token 功能的类。

在第二部分，我们需要构建一个 parser ，该类以一个 lexer 为主要的一个数据成员，对 lexer 解析得到的 Token 进行语法分析树 AST 的构建。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，其实现过程与词法分析有着异曲同工之妙（其本质都是逐渐读取并与相应定义好的模版进行对）。

在实现语法分析器之前，需要首先熟悉 AST.h 文件，该文件定义了相关的语法分析类。

二、AST.h 文件介绍

2.1 class ExprAST

ExprAST 是该文件中定义的基类，代表着语法分析树类的通用内容。该类包含以下几个方面的内容:

（1）枚举类型 ExprASTKind： 用于标识该基类继承后得到的各个派生类的类型。枚举类型包括

Expr_VarDecl, Expr_Return, Expr_Num, Expr_Literal, Expr_Var, Expr_BinOp, Expr_Call, Expr_Print。

这些枚举类型用于标识目前构建的语法分析树是哪一种语法（相当于确定公式的名字）。该基类中有一个私有成员 ExprASTKind kind；kind 中保存的即为语法分析树的类型。

（2）Location location： 用于定位该 AST 的位置。

2.2 NumberExprAST

继承自 ExprAST ，用于构建一个数字的 AST 。

（1）私有成员变量 double val： 用于保存该 AST 的数值。

（2）其他部分：包括构造/析构函数、 getValue（）、classof（）。

2.3 LiteralExprAST

继承自 ExprAST ，用来表示“字面量”，即一个书面表示的数值。

（1）私有成员变量 double val： 用于保存该 AST 的数值。

（2）公有成员变量 std::vector<std::unique_ptr<ExprAST>> values： 用于保存一个 * ExprAST 类型的数组，该数组的每一个元素都指向了一个 ExprAST ，整个数组保存一个字面量。

（3）公有成员变量 std::vector<int64_t> dims

（4）其他部分：包括构造/析构函数、私有成员变量的 get 函数等。

2.4 BinaryExprAST

继承自 ExprAST ，用来表示一个二元运算表达式的 AST。

（1）私有成员变量 char op： 用来存储二元运算符。

（2）私有成员变量 std::unique_ptr<ExprAST> lhs, rhs： 用来存储二元运算的左操作数和右操作数，这两个操作数都是 * ExprAST 类型，指向对应操作数的一个 AST 。

（3）公有成员函数，包括对应的 get 函数等。

2.5 CallExprAST

继承自 ExprAST ，用来表示函数调用的 AST。

（1）私有变量 std::string callee： 保存调用函数名。

（2）私有变量 std::vector<std::unique_ptr<ExprAST>> args： 保存函数调用过程中的参数列表，该数组的每一个元素表示一个指向参数对应 AST 的指针。

（3）公有成员函数，包括对应的 get 函数等。

2.6 PrintExprAST

继承自 ExprAST ，用来表示调用 print 语句的 AST。

（1）私有变量 std::unique_ptr<ExprAST> arg： 表示一个指向 AST 的指针，用来保存 print 的内容。

（2）公有成员函数，包括对应的 get 函数等。

2.7 VariableExprAST

继承自 ExprAST ，用来表示变量命名语句的 AST。

（1）私有变量 std::string name： 表示变量的名字。

（2）公有成员函数，包括对应的 get 函数等。

2.8 VarDeclExprAST

继承自 ExprAST ，用来表示变量定义语句的 AST。

（1）私有变量 std::string name： 表示变量的名字。

（2）私有变量 VarType type： 其中 VarType 是自定义的结构体，保存了一个 std::vector<int64_t> 类型的数组 shape 。

（3）公有成员函数，包括对应的 get 函数等。

2.9 ReturnExprAST

继承自 ExprAST ，用来表示函数返回语句的 AST 。

（1）私有变量 llvm::Optional<std::unique_ptr<ExprAST>> expr： 用来保存一个元组，元组的元素指向返回值的 AST 。

（2）公有成员函数，包括对应的 get 函数等。

2.10 PrototypeAST

该类不是对 ExprAST 的继承，而是用于定义一个函数定义的 AST。该类包含以下内容：

（1）私有变量，包括 location、name、args，其中 args 的生成需要调用前文提到的 VariableExprAST 的语法规则。

（2）相应公有函数。

该类表示一个函数调用中的函数头。

2.11 FunctionAST

（1）私有变量：std::unique_ptr<PrototypeAST> proto 表示一个指向 PrototypeAST 的指针 proto，表示这是函数格式中的函数名及参数表；std::unique_ptr<ExprASTList> body 表示函数大括号中的内容，其中

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

ExprASTList 表示一个数组，每个元素都指向一个 AST 指针（std::vector<std::unique_ptr<ExprAST>>）。

（2）相应公有函数。

```
    if (lexer.getCurToken() != tok_identifier)
      return
parseError<VarDeclExprAST>("identified", "after
variable declaration");
    std::string id(lexer.getId());
    lexer.getNextToken();
```

```
    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));
```

2.12 ModuleAST

只需存储一个 std::vector<FunctionAST> functions 即可。

以上是对 AST.h 文件介绍的全部内容。在这个文件中，完整定义了语法分析器所需要对代码进行的语法分析规则，由小到大，包含了从某些特定的语法AST的实现，到最终所组成的整个 Module 的 AST 的构建方式。

三、Parser.h 文件介绍

此部分主要介绍语法分析器的功能实现，本实验要求实现以下功能：

（1）解析变量的声明，实现成员函数 parserDeclaration()，扩展成员函数 parserType()。要求：

⑴语法变量以 var 开头；

⑵支持三种二维矩阵的初始化

（2）解析函数的常用表达式：

⑴解析标识符，实现成员函数 parserIdentifierExpr()。

⑵解析矩阵二元运算表达式，考虑符号优先级，实现成员函数 parserBinOpRHS()。

以下逐个介绍函数的实现过程。

3.1 parserDeclaration（）

该成员函数用于构建一个变量定义过程的 AST 。对应的语法规则可以描述为：decl ::= var identifier [type] = expr 。

（1）检查变量定义是否以 var 开头：

（2）检查 var 后面的标识符是否是一个合法的变量名。如果是非法，则调用 parseError 类（源代码中已经提供的一个专门用来处理报错信息的类）进行相关报错。

（3）为了支持第三种二维矩阵的表示方法，对该成员函数进行扩展。

完成以上三部分的代码补全后，该成员函数即可正确识别变量定义的开头部分，即 var array …… 后面继续定义的过程需要借助其他成员函数，如“auto expr = parseExpression();”用于读入表达式。

3.2 parserType（）

该部分用于定义特殊的三类格式，用于定义二维矩阵。这里我们主要完成第三种格式定义的拓展。该部分原本的语法分析树可以表示为：

type ::= < shape_list1 > | [shape_list2]
shape_list1 ::= num | num , shape_list1
shape_list2 ::= num] , shape_list2

新增的定义方法为 var a[2][3] = [1, 2, 3, 4, 5, 6];

根据已有的第二种表示方法，改写第三种表示方法。

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

3.3 parserIdentifierExpr（）

该成员函数用于解析标识符。标识符可以分为以下几类：

（1）变量名：返回对应的 AST 即可。

（2）函数调用：调用成员函数 parseExpression（）进行解析，保存在数组元素类型为 *ExprAST 的数组 vector 中，最终返回函数调用的 CallExprAST。

（3）print 语句，检查是否只有一个参数，若合法则返回 print 函数对应的 PrintExprAST。

以下逐个介绍。

（1）如果读入标识符（保存在std::string name(lexer.getId());中）之后的下一个读入不是“（”，说明这不是一个函数调用，而是普通变量的定义。

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

（2）如果读入标识符后发现是函数调用，则首先创建一个 args 用于保存参数列表。并不断读入，直到遇到右括号为止。

```
lexer.consume(Token('('));
    std::vector<std::unique_ptr<ExprAST> > args;
    if (lexer.getCurToken() != ')')
    {
      while(1)
      {
        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 args");
        lexer.getNextToken();
      }
    }
    lexer.consume(Token(')'));
```

在参数列表的保存中，每遇到新的参数，就调用成员函数 parserExpression（）进行解析，并保存在 vector 中。遇到非法输入（参数之间没有逗号隔开、逗号后面紧跟括号等）则会调用报错类。

（3）print 语句解析，当 name 为 print 时检查参数列表即可。

```
    if (name == "print")
    {
      if (args.size() != 1)
        return parseError<ExprAST>("<single
arg>", "as argument to print()");
    }
```

最后返回。

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

3.4 parserBinOpRHS（）

本成员函数主要用于处理关于矩阵的二元运算表达式。在前面的代码中，定义了基本运算符号的优先级：

“+、-”：20；　　“*”：40；

同时为了保证从左向右依次运算的属性，在实现该成员函数时，要求每读入一个新的运算数，都要将之前的算符优先级自增。以下是基本的代码展示。

```
  std::unique_ptr<ExprAST> parseBinOpRHS(int
exprPrec, std::unique_ptr<ExprAST> lhs) {
    while(1)
    {
      int prec = getTokPrecedence();
      if (prec < exprPrec)
        return lhs;
      int opt = lexer.getCurToken();
      lexer.consume(Token(opt));
      auto loc = lexer.getLastLocation();

      auto rhs = parsePrimary();
      if (!rhs)
        return
parseError<ExprAST>("expression","to end
operation");
      int r_prec = getTokPrecedence();
      if (prec < r_prec)
      {
        rhs = parseBinOpRHS(prec + 1,
std::move(rhs));
        if (!rhs)
          return nullptr;
      }
      lhs =
std::make_unique<BinaryExprAST>(std::move(loc),
opt, std::move(lhs), std::move(rhs));
    }
  }
```

函数的主体部分是一个 while 循环，该循环在 rhs 读入结束后返回。函数定义了左运算数 lhs 和右运算数 rhs 。当读入的运算符优先级更低时，将运算符左边的表达式计算后保存到 lhs 中。若读入一个更高优先级

的运算符，则调用函数本身，将 rhs 右侧进行运算，并更新当前的 rhs 。最终完成整个算术表达式的解析后，返回 lhs 。

四、实验验证

完成以上内容后，运行测试用例 test_8 到 test_12。以下是终端中的执行结果。

```
parallels@ubuntu-linux-22-04-desktop:~/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
        Call 'print' [ @../test/test_8.pony:8:3
          var: a @../test/test_8.pony:8:9
        ]
      } // Block
```

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_9.pony -emit=ast
Parse error (8, 13): expected '<single arg>' as argument to print() but has Toke
n 59 ';'
Parse error (8, 13): expected 'nothing' at end of module but has Token 59 ';'
```

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony .
/test/test_10.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_10.pony:4:1
      Params: []
      Block {
        VarDecl a<> @../test/test_10.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_10.pony:6:11
        VarDecl b<2, 3> @../test/test_10.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_10.pony:7:17
        VarDecl d<2, 3> @../test/test_10.pony:8:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00] @../test/test_10.pony:8:17
        Call 'print' [ @../test/test_10.pony:9:3
          var: b @../test/test_10.pony:9:9
        ]
      } // Block
```

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ../test/test_11.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_11.pony:4:1
      Params: []
      Block {
        VarDecl a<> @../test/test_11.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_11.pony:6:11
        VarDecl b<2, 3> @../test/test_11.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_11.pony:7:17
        VarDecl c<> @../test/test_11.pony:8:3
          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_11.pony:8:11
        VarDecl d<2, 3> @../test/test_11.pony:9:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @../test/
test_11.pony:9:17
        VarDecl e<> @../test/test_11.pony:10:3
          BinOp: * @../test/test_11.pony:10:17
            BinOp: + @../test/test_11.pony:10:14
              var: a @../test/test_11.pony:10:12
              var: c @../test/test_11.pony:10:14
            BinOp: + @../test/test_11.pony:10:20
              var: b @../test/test_11.pony:10:18
              var: d @../test/test_11.pony:10:20
        Call 'print' [ @../test/test_11.pony:11:3
          var: e @../test/test_11.pony:11:9
        ]
      } // Block
```

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ../test/test_12.pony -emit=ast
  Module:
    Function
      Proto 'multiply_transpose' @../test/test_12.pony:3:1
      Params: [a, b]
      Block {
        Return
          BinOp: * @../test/test_12.pony:4:25
            Call 'transpose' [ @../test/test_12.pony:4:10
              var: a @../test/test_12.pony:4:20
            ]
            Call 'transpose' [ @../test/test_12.pony:4:25
              var: b @../test/test_12.pony:4:35
            ]
      } // Block
    Function
      Proto 'main' @../test/test_12.pony:7:1
      Params: []
      Block {
        VarDecl a<> @../test/test_12.pony:8:3
          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_12.pony:8:11
        VarDecl b<2, 3> @../test/test_12.pony:9:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @../test/
test_12.pony:9:17
        VarDecl c<> @../test/test_12.pony:10:3
          Call 'multiply_transpose' [ @../test/test_12.pony:10:11
            var: a @../test/test_12.pony:10:30
            var: b @../test/test_12.pony:10:33
          ]
        VarDecl d<> @../test/test_12.pony:11:3
          Call 'multiply_transpose' [ @../test/test_12.pony:11:11
            var: a @../test/test_12.pony:11:30
            var: b @../test/test_12.pony:11:33
          ]
        VarDecl e<> @../test/test_12.pony:12:3
          BinOp: + @../test/test_12.pony:12:39
            BinOp: + @../test/test_12.pony:12:26
              BinOp: * @../test/test_12.pony:12:24
                Call 'transpose' [ @../test/test_12.pony:12:11
                  var: a @../test/test_12.pony:12:21
                ]
                var: c @../test/test_12.pony:12:24
              Call 'transpose' [ @../test/test_12.pony:12:26
                var: b @../test/test_12.pony:12:36
              ]
            var: d @../test/test_12.pony:12:39
        Call 'print' [ @../test/test_12.pony:13:3
          var: e @../test/test_12.pony:13:9
        ]
      } // Block
```