MLIR 编译框架的使用与探索
第一部分：词法分析

Junhao Dai daijunhao@sjtu.edu.cn

April 23, 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。

在第一部分中，我们需要构建一个词法分析器来识别 Pony 语言中的各个词法单元（Token），包括关键字（如 var、def 和 return）、特殊符号、数字以及变量名/函数名等。我们需要通过相关的函数来获取 Token，并对其判断合法性，同时针对非法格式输出响应的报错信息。

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 getNextChar()

该函数从 curLineBuffer 中获取当前行的下一个 char，如果已经处理到当前行最后一个 char，则通过读取下一行来更新 curLineBuffer 以确保 curLineBuffer 非空。核心代码如下：

```
int getNextChar() {
    if (curLineBufer.empty())
      return EOF;

    auto nextChar = curLineBuffer.front();
    curLineBuffer = curLineBuffer.drop_front();
    curCol++;

    if (curLineBuffer.empty()) {
      curLineNum++;
      curCol = 0;
```

```
12        curLineBuffer = readNextLine();
13      }
14
15      return nextChar;
16    }
```

2.2 成员函数 getTok()

在此函数中，我们需要实现对于一个 Token 的读取和分析。getTok() 函数识别标识符的要求如下：

- 能够识别 “return”、“def” 和 “var” 三个关键字；
- 能够识别标识符（函数名，变量名等）：
- 标识符应以字母开头；
- 标识符由字母、数字或下划线组成；
- 按照使用习惯，标识符中不能出现连续的下划线；
- 按照使用习惯，要求标识符中有数字时，数字须位于标识符末尾；
- 在识别每种 Token 的同时，将其存放在某种数据结构中，以便最终在终端输出

getTok() 函数还需要改进识别数字的方法，使编译器可以识别并在终端报告非法数字，非法表示包括：9.9.9，9..9，.999，..9，9.. 等。此处我们定义两种非法输入，一种是数字中含有多个小数点，另一种是数字的小数点位于开头或者末尾。实际情况中这两种错误可能同时存在，如..9，9.. 等，输出报错信息时将输出检测到的第一种，而不会全部输出。

对于以上要求，函数将首先判断是读入一个数字或者是字符，若为数字则进入识别数字判断，反之则进入识别字符判断。数字判断中，使用 `dot_count` 对小数点数量计数，最终得到的数字字符串判断其开头和结尾是否是小数点。实现的核心代码如下：

```
if (isdigit(lastChar) || lastChar == '.') {
  std::string numStr;
  int dot_count = 0;
  do {
    if (lastChar == '.')
      dot_count++;
    numStr += lastChar;
    lastChar = Token(getNextChar());
  } while (isdigit(lastChar) || lastChar == '.');

  if (numStr.back() == '.' || numStr.front() == '.') {
    std::cerr
        << "Error: Invalid number at line " << curLineNum << " column "
        << curCol
        << " :the decimal point is at the beginning or end of the number"
        << std::endl;
    return Token(0);
```

```
  }

  if (dot_count > 1) {
    std::cerr << "Error: Invalid number at line " << curLineNum
              << " column " << curCol << " :multiple decimal points"
              << std::endl;
    return Token(0);
  }

  numVal = strtod(numStr.c_str(), nullptr);
  return tok_number;
}
```

而对于字符识别，函数中定义了多个布尔型变量来记录错误类型，这样便于读入整个错误字符后返回报错并丢弃当前错误字符，输出中将使用 ERROR_TOKEN 来代替该错误字符。主要分为：连续下划线错误、以下划线开头错误、数字不在末尾错误。核心代码如下所示：

```
if (isalpha(lastChar) || lastChar == '_') {
  identifierStr = lastChar;
  bool underline_in_row = false;
  bool digit_in_middle = false;
  bool digit_in_middle_error = false;
  bool underline_in_row_error = false;
  bool start_with_underline_error = false;

  if (lastChar == '_') {
    std::cerr << "Error: Invalid identifier at line " << curLineNum
              << " column " << curCol
              << " :identifier starts with underline" << std::endl;
    start_with_underline_error = true;
  }

  while (isalnum((lastChar = Token(getNextChar()))) || lastChar == '_') {
    identifierStr += lastChar;
    if (isdigit(lastChar))
      digit_in_middle = true;
    if (lastChar == '_' && underline_in_row) {
      std::cerr << "Error: Invalid identifier at line " << curLineNum
                << " column " << curCol << " :multiple underline in a row"
                << std::endl;
      underline_in_row_error = true;
    }

    if (lastChar == '_')
      underline_in_row = true;
    else
```

```
      underline_in_row = false;

    if (digit_in_middle && isalpha(lastChar)) {
      std::cerr << "Error: Invalid identifier at line " << curLineNum
                << " column " << curCol
                << " :digit in the middle of the identifier" << std::endl;
      digit_in_middle_error = true;
    }
  }

  if (identifierStr == "return")
    return tok_return;
  if (identifierStr == "def")
    return tok_def;
  if (identifierStr == "var")
    return tok_var;

  if (underline_in_row_error || start_with_underline_error ||
      digit_in_middle_error)
    return Token(0);
  return tok_identifier;
}
```

2.3 成员函数 dumpToken()

在此函数中，我们需要使用 lexer 遍历整个文档，最终按顺序输出识别到的每一种 Token。本次的输出格式为：首先输出全部的报错信息，其次输出读取到的全部 token，每个 token 用空格隔开，非法的 token 使用 "ERROR_TOKEN" 来代替。因此我们需要使用一个 vector<string> 数组来存储所有的 token 的字符串形式便于最后统一输出，每次调用 getCurToken() 之后需要判断 token 类型，并将其转换为合适的字符串形式。核心代码实现如下：

```
std::vector<std::string> tokens;

while (lexer.getNextToken() != pony::tok_eof) {
if (lexer.getCurToken() == tok_identifier) {
  tokens.push_back(lexer.getId().str());
} else if (lexer.getCurToken() == pony::tok_number) {
  std::string numStr = std::to_string(num);
  numStr.erase(numStr.find_last_not_of('0') + 1, std::string::npos);
  tokens.push_back(numStr);
} else if (lexer.getCurToken() == pony::tok_def) {
  tokens.push_back("def");
} else if (lexer.getCurToken() == pony::tok_var) {
  tokens.push_back("var");
} else if (lexer.getCurToken() == pony::tok_return) {
  tokens.push_back("return");
```

```
} else if (lexer.getCurToken() == pony::tok_eof) {
  continue;
} else if (lexer.getCurToken() == ';' || lexer.getCurToken() == '(' ||
           lexer.getCurToken() == ')' || lexer.getCurToken() == '{' ||
           lexer.getCurToken() == '}' || lexer.getCurToken() == '[' ||
           lexer.getCurToken() == ']' || lexer.getCurToken() == ',') {
  tokens.push_back(std::string(1, lexer.getCurToken()));
} else if (lexer.getCurToken() == 0) {
  tokens.push_back("ERROR_TOKEN");
}
}

for (auto &token : tokens) {
std::cout << token << " ";
}
std::cout << std::endl;
```

3 实验结果

在对词法分析器构建完毕后，可以通过运行测试用例 test_1 至 test_7 来检查词法分析器的正确性。测试结果如图1所示：

```
[100%] Built target pony
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_1.pony -emit=token
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , 2 , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_2.pony -emit=token
Error: Invalid identifier at line 9 column 9 :multiple underline in a row
def multiply_transpose ( a , b ) { return transpose ( a ) transpose ( b ) ; } def main ( ) { var ERROR_TOKEN
 [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ] ; var b 2 , 3 [ 1 , 2 , 3 , 4 , 5 , 6 ] ; var c multiply_transpose ( a ,
b ) ; print ( c ) ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_3.pony -emit=token
Error: Invalid identifier at line 5 column 10 :digit in the middle of the identifier
def main ( ) { var ERROR_TOKEN [ 2 ] [ 3 ] [ 1 , 2 , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_4.pony -emit=token
Error: Invalid identifier at line 5 column 9 :identifier starts with underline
def main ( ) { var ERROR_TOKEN [ 2 ] [ 3 ] [ 1 , 2 , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_5.pony -emit=token
Error: Invalid number at line 5 column 26 :multiple decimal points
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , ERROR_TOKEN , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_6.pony -emit=token
Error: Invalid number at line 6 column 26 :the decimal point is at the beginning or end of the number
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , ERROR_TOKEN , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_7.pony -emit=token
Error: Invalid number at line 5 column 25 :the decimal point is at the beginning or end of the number
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , ERROR_TOKEN , 3 , 4 , 5 , 6 ] ; }
```

Figure 1: Test Result form test_1 to test_7

从结果可以看到，除 test_1 以外的测试样例均有输入非法的标识符或数字，test_2 中由于标识符含有连续的下划线因此报错；test_3 中数字不在标识符的末尾因此报错；test_4 中因标识符以

下划线开头而报错；test_5 中因数字含有多个小数点而报错；test_6 和 test_7 均因小数点在数字的开头或者末尾处而报错。以上为 Part 1 部分的测试结果，结果表明已实现基本的词法分析功能。

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。由于 MLIR 和 LLVM 的安装以及环境配置过于复杂，本次实验提供了一个 Docker 镜像作为实验平台，这极大降低了初学者的学习成本，减少了不必要的配置时间，避免了过多耗费尽力在环境配置上，而使得大家可以专心研究核心部分的任务，因此再次向老师以及助教表示感谢。

除此以外，实验书中对于 Docker 的配置并不是特别清晰，许多同学对于 REPO_PATH 感到疑惑，因为参数错误导致 docker 容器的挂载、创建、运行、退出等操作失误，而实验书中仅有官方文档的链接，我也是通过与同学讨论，找出了自己命令的参数错误从而逐步学会使用，如果可以的话希望以后能在这一处有非官方文档外的总结文档供入门同学参考。

参考文献

[1] Docker 101 Tutorial https://www.docker.com/101-tutorial/

[2] Docker 学习笔记 创建容器、退出容器、查看容器、进入容器、停止容器、启动容器、删除容器、查看容器详细信息

[3] Docker-从入门到实践 https://yeasy.gitbook.io/docker$_{p}ractice/container/run$