CS2301 《编译原理》大作业

第一部分-词法分析

常烁晨

521021910369

一、实验背景简介

1.1、编译框架介绍：

本次编译原理大作业使用MLIR编译框架。MLIR 的全称为 Multi-Level Intermediate Representation，即“多层中间表示”，该框架将编译器的基础结构进行整合与拓展，通过中间模态的表示方法整合现有的编译器相关工作，提供了一种构建可重用和可扩展编译器基础结构的新方法。

1.2、Pony语言介绍：

本次大作业基于Pony语言进行编译，目标是完成一个小型的编译器，对Pony语言进行解析。Pony 语言是一种自定义的、基于张量(tensor)的语言。支持少量功能，包括函数定义、基本数学运算以及打印功能。目前 Pony 语言支持一维和二维张量以及 64 位浮点数 (C 语言中的 double) 数据类型。

1.3、大作业任务介绍：

本次大作业要求基于MLIR编译框架，搭建一个简单的编译器，使其可以编译基于 Pony 语言的代码。 本次实验报告实现的是大作业的第一部分：词法分析。

在该阶段，我们要构建一个词法分析器来识别 Pony 语言中的各种词法单元 (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。我们要通过相关函数来获取 Token，判断其合法性，若test文件代码合法，将在终端按序输出合法的Token，若文件中存在非法字符，则在终端针对非法格式输出相应的错误信息。

1.4、实验环境搭建：

本次实验我使用的实验环境基于Macbook Air M1（arm64处理器）的Parallels Desktop虚拟机，安装发行版Linux环境Ubuntu 22.04。

（1）首先在虚拟Linux环境中使用安装前置工具，包括：git, cmake, clang, lld, ninja 。

（2）通过git工具，安装MLIR环境。

注意在M1芯片的MacBook上执行命令时，

-DLLVM_TARGETS_TO_BUILD="X86;NVPTX;AMDGPU" 要更改为
-DLLVM_TARGETS_TO_BUILD="AArch64;NVPTX;AMDGPU"。

（3）安装完成后，通过cmake —build指令编译MLIR。编译过程由于不同电脑性能的差异，编译时间可能会在几十分钟至一到两小时不等。

（4）在主目录下载pony_compiler 。通过git克隆相应的仓库到本地的主目录，并在pony_compiler/build 文件夹中运行cmake -G Ninja 指令，进行相应文件配置文件到编译。注意，在此处路径填写时，如果终端报错显示无法找到对应的文件路径，可以尝试将实验指导书中

**LLVM_DIR=/Path/to/llvm-project/build/lib/cmake/llvm **
**MLIR_DIR=/Path/to/llvm-project/build/lib/cmake/mlir ** 修改为

**export LLVM_DIR=/Path/to/llvm-project/build/lib/cmake/llvm **
**export MLIR_DIR=/Path/to/llvm-project/build/lib/cmake/mlir **

猜测该问题出现的原因可能是ubuntu版本的差异（如20.04和22.04）或者电脑型号的差异（如windows电脑与m系列芯片的mac电脑）。

二、实验要求

2.1、词法分析功能实现

文件地址: **/pony_compiler/pony/include/pony/Lexer.h**

要求实现以下功能:

（1） 实现成员函数 getNextChar()

a.注意几种corner case的特殊处理，比如读到某行结尾，读到文档结尾等情况。

b.注意行列等位置信息的更新。

（2）补充成员函数 getTok()

a.能够识别“return”、“def”和“var”三个关键字

b.能够识别标识符:

- 标识符以字母或下划线开头
- 标识符由字母、数字和下划线组成

• 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。例如:有效的标识符可以是 a123, b_4, placeholder 等。

c.改进识别number的方法，使编译器可以识别并在终端报告非法number，非法表示包括:9.9.9， 9..9，.123等。

2.2 词法分析验证程序实现

文件地址: **/pony_compiler/pony/ponyc.cpp + /pony_compiler/pony/include/pony/Lexer.h**

要求实现以下功能:

（1）补充ponyc.cpp文件中“词法分析器正确性”验证程序int dumpToken()。

（2）扩展Lexer.h文件中getTok()函数，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。

a.输入文件为Pony语言定义的函数等。

b.如果词法分析器没有识别出错误，则按顺序输出识别到的Token。

c.如果词法分析器识别出错误，则在终端输出详细的错误信息

d.输出信息的具体形式可参考接下来第四部分的测试示例（test_1 到 test_7)。

三、代码编写

3.1 Lexer.h文件

（1）实现成员函数getNextChar()

```
int getNextChar() {
  if (curLineBuffer.empty())
    return EOF;
  curCol++;
  auto thechar = curLineBuffer.front();
  curLineBuffer = curLineBuffer.drop_front();
  if (curLineBuffer.empty())
  {
    curLineBuffer = readNextLine();
    curLineNum++;
  }
```

```
    if (thechar == '\n')
    {
      curCol = 0;
    }
    return thechar;
  }
```

int getNextChar() 函数是 lexer 类中的私有成员函数，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。其中 curLineBuffer 是 lexer 类中的一个 llvm::StringRef 类型的私有成员变量，该类型与 std 中的 string 类型类似，支持对字符串进行操作。在 lexer 中被用作读取字符的缓冲区。

函数 getNextChar() 通过读取缓冲区 curLineBuffer 中的字符直到读取到文件结束的标识符EOF。缓冲区一次性会将源文件中的一行代码整体以字符串的形式读入，当 getNextChar() 函数读取到换行符 '\n' 时，将会调用 readNextLine() 函数用于读取下一行代码，并将当前的行数加1，列数归零。其余情况下函数每读取一个字符时，都会自动更新当前的行列值，保存在 lexer 类的成员对象 curCol 和 curLineNum 中，并返回 int 类型的ascii码来表示当前读取的字符。

（2）补充成员函数 getTok()

```
    if (isalpha(lastChar) || lastChar == '_')
    {
      std::string varStr;
      bool isError = false, isDigit = false;
      do{
        if(isdigit(lastChar))
          isDigit = true;
        if(isDigit)
        {
          if(isalpha(lastChar) || lastChar == '_')
          {
            isError = true;
            flag = false;
            errorLocation.push_back(lastLocation);
            errorType.push_back(1);//1-type:number not ending
            break;
          }
        }
        varStr += lastChar;
        lastChar = Token(getNextChar());
      }
```

```
      while(isalnum(lastChar) || lastChar == '_');
      if(!isError)
      {
        if(varStr == "return")
          return tok_return;
        if(varStr == "def")
          return tok_def;
        if(varStr == "var")
          return tok_var;
        identifierStr = varStr;
        return tok_identifier;
      }
    }
```

第一步是编写对 Pony 语言的关键字以及标识符（包括函数名、变量名等）的识别。在此之前首先要介绍 lexer 类中的 Token 枚举类型。

```
enum Token : int {
  tok_semicolon = ';',
  tok_parenthese_open = '(',
  tok_parenthese_close = ')',
  tok_bracket_open = '{',
  tok_bracket_close = '}',
  tok_sbracket_open = '[',
  tok_sbracket_close = ']',

  tok_eof = -1,

  tok_return = -2,
  tok_var = -3,
  tok_def = -4,

  tok_identifier = -5,
  tok_number = -6,
};
```

由此可见，编译器中保存 Token 的底层类型为 int 型，其中 ‘;’ ‘(‘ ‘)’ ‘[‘ ‘]’ ‘{‘ ‘}’ 等字符类型直接保存为 ascii 码，而其余的关键字或标识符等，则用负整数（-1 ~-6）来保存。

getToken() 函数返回一个 Token 枚举类型，第一部分实现了该函数读取一个关键字或者标识符的功能。首先，Pony 语言中的标识符（包括变量名和函数名等）的格式要求为：

a. 以字母或下划线开头 b. 由数字、字母和下划线组成 c.如果有数字，则数字都在末尾

可以发现，编译器要求识别出的三个关键字，同样符合标识符的要求，因此该函数可以同时将标识符和关键字进行识别。

要完成函数的补充，需要用到上一步我们编写的 getNextChar 函数。事实上 getNextChar 通过逐个读取字符，就能实现关键字的读入，因此这一部分主要完成的是读入字符串的识别、匹配分类以及非法输入的报错。第一部分识别关键字和标识符，主体部分通过一个 do-while 循环，逐个读取字符，直到字符不是数字、字母和下划线中的一个为止。若输入合法，函数将会完整读入并将其保存在字符串中。函数中设置了 bool 类型的变量用于监测目前为止输入字符的种类。如 isDigit 监控目前这一可能的标识符中是否包含了数字的输入。如果输入数字，该布尔变量置 1 ，如果在输入数字之后又输入了字母或者下划线，则说明该标识符非法，将 bool 变量 isError 置 1 ，并通过vector记录这一非法信息（具体实现在后文中展示）并跳出循环。

当完成一个合法的标识符（或关键字）的读入（ isError 保持为 0 ）后，首先查验这是不是一个关键字。通过字符串的比较，如果输入的是一个关键字，则返回 tok_return , tok_def , tok_var 中的一种，否则说明读取到的是一个标识符，将保存的字符串存储在 lexer 类中的 identifierStr 中，并返回 tok_identifier（表示输入为标识符）。

接下来实现对数字的读取。

```
if (isdigit(lastChar) || lastChar == '.')
{
  std::string numStr;
  bool isError = false, isRadix = false, isDigit = false;
  do
  {
    if(isdigit(lastChar))
      isDigit = true;
    if(lastChar == '.')
      isRadix = true;
```

```
        if ((!isDigit) && isRadix)
        {
          isError = true;
          flag = false;
          errorLocation.push_back(lastLocation);
          errorType.push_back(2);//2-type: .begin
          break;
        }
        if(isRadix)
        {
          if(lastChar == '.')
          {
            flag = false;
            errorLocation.push_back(lastLocation);
            errorType.push_back(3);//3-type: more than
one .
            isError = true;
            break;
          }
        }
        numStr += lastChar;
        lastChar = Token(getNextChar());
      } while (isdigit(lastChar) || lastChar == '.');
      if(!isError)
      {
        numVal = strtod(numStr.c_str(), nullptr);
        return tok_number;
      }
```

第二部分补充 getTok() 函数，改进识别数字的方法，使得编译器可以识别并且在终端中报告非法数字，非法表示包括9.9.9，9..9，.123 等。这一部分我将具体的错误分为两类，一个是以小数点开头的输入，另一类是出现多个小数点。

类似于标识符的识别过程，在函数体中设置了 bool 类型变量，isRadix 监控当前是否有小数点读入，isDigit 监控当前是否有数字读入。函数主体部分由一个 do-while 循环实现，不断通过 getNextChar() 读入字符，若输入始终合法，则循环不断执行，直到读入了一个非数字、非小数点的字符后循环结束。此时字符串中保存着一个合法的数字。

由于每读入一个字符，布尔变量都会监视截至目前的读入是否合法，如果出现非法读入，则循环终止，并根据 bool 变量监控的内容判断是哪一种非法输入。如当目前的 isDisit 为 0 时表明目前尚无数字的读入，此时若 isRadix 为 1 ，说明读入的非法数字是以小数点开头的，将当前类 lexer 中的 location 信息以及错误类型保存在 vector 中。如当前 isRadix 为 1 ，表明目前为止已经读入了小数点，此时检查后续输入中如果有小数点读入，则说明当前输入非法的原因是出现了多于一个小数点。

如果最终输入合法，则将输入的字符串转化为 double 类型，保存在类的成员变量 numVal 中，并返回 tok_number 表明读入的是一个实数。

最后是 Lexer.h 文件中对非法输入的处理手段。由前面的介绍，在处理非法输入时是通过 bool 变量来监控，因此在 lexer 类中设置一个私有的成员变量 flag 用于标识 lexer 遍历文件的过程中是否遇到了非法输入。以下展示新增的代码。

```
private:

  bool flag = true;
  std::vector<Location> errorLocation;
  std::vector<int> errorType;
```

将 lexer 类新增了三个私有的成员变量，其中变量 flag 用于标识 lexer 遍历文件的过程中是否遇到了非法输入，该变量在类创建时被初始化为 true ，一旦在读取过程中遇到非法输入（即函数 getTok() 中 isError 置 1 后， flag 立即置为 false 并且不再修改。这一策略可以参照我们日常使用的编译器的处理手段，当代码中出现问题后，编译器将不会输出运行结果，而是将所有的错误逐一输出。

同时新增了两个私有的 vector 类型为 errorLocation 和 errorType 。这两个容器用于按顺序保存输入文件中的非法输入，其中 errorLocation 是一个 Location 类的容器，用于保存非法输入的位置。lexer 类中有私有成员变量 lastLocation ，在每次调用 getTok() 时保存当前读入的位置。在非法输入处理时，当函数 getTok() 检测到错误时，便会将当前的位置保存在 errorLocation 中，将当前的错误类型保存在 errorType 中，其中 errorType[i] = 1 代表标识符的数字后面出现了字母或者下划线，errorType[i] = 2 代表数字以小数点开头， errorType[i] = 3 代表一个数字和小数点组成的串中有不止一个小数点。

由于新增的成员变量属于私有类型，因此要对类编写相应的公有成员函数，便于函数调用 lexer 类，在终端中进行报错。

公有成员函数的编写如下所示。

```
public:
  bool getFlag() { return flag; }
  void errorIndicate()
  {
    assert(flag == false);
    for(int i=0; i<errorType.size(); i++)
    {
      std::cout <<"<"<<i+1<<"> "<< "error located at(" <<
errorLocation[i].line <<", "<<errorLocation[i].col <<"): ";
      switch(errorType[i])
      {
          case 1:
          std::cout<<"digits should be at the end of an
identifier"<<'\n';
          break;
        case 2:
          std::cout<<"missing digits before the decimal
point"<<'\n';
          break;
        case 3:
          std::cout<<"containing more than one dicimal
point"<<'\n';
          break;
      }
    }
  }
```

该公有函数用于外部函数调用时，如果判断当前输入的测试文件代码中有非法输入，则逐个输出报错信息。

3.2 ponyc.cpp 文件

该文件中所需要编写的内容为 int dumpToken() 函数。在该函数中，测试文件创建相应的 lexer 类，并对测试文件进行遍历。如果没有非法输入，则逐个输出读取到的 Token ，否则逐条输出错误信息。

补充代码如下。

```
int dumpToken() {
/*                    code                      */
  auto buf = fileOrErr.get()->getBuffer();
  LexerBuffer lex(buf.begin(), buf.end(),
std::string(inputFilename));
  lex.getNextToken();
  bool isflag = true;
  while(lex.getCurToken() != tok_eof)
  {
    if(!lex.getFlag())
      isflag = false;
    lex.getNextToken();
  }
  if(!isflag)
  {
    lex.errorIndicate();
    return 0;
  }
  auto buffer = fileOrErr.get()->getBuffer();
  LexerBuffer lexer(buffer.begin(), buffer.end(),
std::string(inputFilename));
  lexer.getNextToken(); // prime the lexer
  while (lexer.getCurToken() != tok_eof)
  {
    switch(lexer.getCurToken())
    {
      case tok_return:
        std::cout << "return" << " ";
        break;
      case tok_var:
        std::cout << "var" << " ";
        break;
      case tok_def:
        std::cout << "def" << " ";
        break;
      case tok_identifier:
        std::cout << std::string(lexer.getId()) << " ";
        break;
      case tok_number:
        std::cout << lexer.getValue() << " ";
        break;
      default:
        std::cout << char(lexer.getCurToken()) << " ";
    }
    lexer.getNextToken();
   }
  return 0;
}
```

在 dumpToken() 函数中对输入的测试代码进行遍历。首先新建一个 LexerBuffer 类的对象 lex ，函数中使用的 LexerBuffer 由我们之前编写的 lexer 类继承而来，在子类中重写的基类的虚函数（用于缓冲区换行，将测试代码读入类中）。

为了检测输入的测试代码中是否含有非法输入，首先仿照代码中初始化的方式对 lex 对象进行初始化，之后通过一个 while 循环遍历测试代码直到读取到 EOF 。由于先前我们在类 lexer 中添加了对象 flag ，因此只需在遍历完成后检测 lex 的 flag 值，就可以确定测试代码之中是否含有非法输入。

如果输入测试代码含有非法输入，由于 lex 在遍历代码的过程中已经将相关信息保存在类的两个 vector 容器之中，因此调用相应的公有函数时，即可将错误信息输出，打印在终端中。

如果遍历结束后未检测到非法输入，则新建一个 LexerBuffer 的对象 lexer，仿照前面的代码对其进行初始化并遍历。由于测试代码输入全部合法，因此只需在遍历的过程（while 循环实现遍历）中，每调用 getTok() 函数获得一个 Token 的输入，便在终端中将其打印。根据函数的返回类型，可以确定读入的字符种类。如果读入了关键字，则将该关键字打印，如果读入标识符或者数字，则将对应的成员变量进行打印，如果读入单个字符的枚举类型，如 ‘;’等，则根据保存的 ascii 码，将字符还原并打印。

四、实验验证

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。

首先在主目录下打开终端，进入主目录下的/pony_compiler/build 文件中，输入以下指令：

cmake --build . --target pony

将pony文件构建完成后，可以输入对应的测试文件路径，将测试代码读入到相应对象中进行编译。如要使用词法分析器分析 text_1 ，可以输入以下指令：

../build/bin/pony ../test/test_1.pony -emit=token

以下展示 test_1 到 test_7 的终端输出结果：

test_1:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_1.pony -emit=token
def main ( ) { var a [ 2 ] [ 3 ] = [ 1 , 2 , 3 , 4 , 5 , 6 ] ; } parallels@ubunt
```

test_2:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_2.pony -emit=token
def multiply_transpose ( a , b ) { return transpose ( a ) * transpose ( b ) ; }
def main ( ) { var a = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ] ; var b < 2 , 3 > = [ 1
 , 2 , 3 , 4 , 5 , 6 ] ; var c = multiply_transpose ( a , b ) ; print ( c ) ; }
```

test_3:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_3.pony -emit=token
<1> error located at(5, 8): digits should be at the end of an identifier
```

test_4:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_4.pony -emit=token
def main ( ) { var _dd4 [ 2 ] [ 3 ] = [ 1 , 2 , 3 , 4 , 5 , 6 ] ; } parallels@ub
```

test_5:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_5.pony -emit=token
<1> error located at(5, 22): containing more than one dicimal point
<2> error located at(5, 24): missing digits before the decimal point
```

test_6:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_6.pony -emit=token
<1> error located at(6, 22): containing more than one dicimal point
<2> error located at(6, 24): containing more than one dicimal point
```

test_7:

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_7.pony -emit=token
<1> error located at(5, 22): missing digits before the decimal point
```

五、优化方案、总结与反思、致谢

在完成代码的编写、调试，项目的构建，test文件的测试，以及上述实验报告的编写后，我大致了解了如何借助课上所学的词法分析器的原理，对一个简单的 Pony 语言输入进行词法分析，初步了解了现代编程过程中现代编译器的最初一步的编译方式，加深了我对编译器 “翻译” 源代码到目标代码这一过程的理解。

由于这一部分只是构建了一个简单的词法分析器，距离真正的大型语言编译过程还有很远，因此我才得以通过自己的努力完成这部分代码的编写，以及实验报告的相关分析。

完成第一部分的作业要求后，我反思了自己编写代码的部分缺漏：

（1）编译器的编译效率比较低、封装不完美。

由于我在编写代码的时候创建了两个 LexerBuffer 类的对象，如果在输入合法的情况下将需要遍历两次代码，首先遍历一遍代码检查是否存在非法输入，之后再次遍历代码将 Token 逐个输出，这使得在代码较长的话，该词法分析器将要用更多的时间。

出现这一状况的根本原因是我的代码封装度不够。由于我只在错误处理的部分应用了两个 vector 容器存储错误信息，而没有想到在类中新建一个 Token 类的 vector 容器逐个保存读入的 Token ，并创建相应的公有函数适时将 vector<Token> 进行遍历输出，这导致我需要提前扫描输入代码，检查是否存在非法输入，并再次遍历逐个输出 Token 。

同样由于代码欠缺封装，在 ponyc.cpp 文件中暴露了过多 LexerBuffer 类的底层数据内容（如通过 while 循环，逐个输出 getToken 函数的返回值，等），这一点做的不够完美。

（2）终端的报错内容不够有针对性。

现代编译器报错信息十分完善，如可以输出源代码中含有非法输入的行，并用相应的符号标记出非法输入的位置，同时相比之前其报错信息也更加完善和有针对性。在通过测试样例时，我发现许多非法输入会经过两次报错，即分别对 “小数点前没有数字” 和 “数字中有多于一个小数点” 进行报错，实际应用过程中，这一报错信息可以进行适当的整合。

上面提到的不足之处，以及老师和助教学长之后的反馈和建议，我会在大作业第三部分代码优化中进行修改和完善。

最后，感谢赵杰茹老师的授课讲解以及在群内的答疑，感谢张炜创学长一周以来帮我解决环境配置过程中的问题，感谢互联网上各类程序网站对于各种报错内容及其相应处理手段的记录，感谢水源社区的源友对于Linux环境引用目录问题的相关解答！