

Project 1 Report

Name: Fanyue Zhu Student id: 522031910547

1 Objectives

Build a lexical analyser to identify the various lexical units (tokens) in the Pony language, including keywords (such as var, def and return), special symbols, numbers and variable/function names. The Token is obtained through a function that determines its legitimacy and outputs an error message for illegal formats.

2 Implementation

2.1 Lexical Analysis Function Implementation

2.1.1 getNextChar()

First determine if it is an end of line. If not, directly return to the next character; otherwise, read the next line and update the rows and columns, if the next line is empty then the proof is EOF, otherwise return to the normal first character.

```
1  int getNextChar()
2  {
3      if (curCol == curLineBuffer.size()) {
4          curLineBuffer = readNextLine();
5          curCol = 0;
6          curLineNum++;
7          if (curLineBuffer.empty()) {
8              return EOF;
9          }
10     }
11     return curLineBuffer[curCol++];
12 }
```

2.1.2 getTok()

1. Keyword Identification: As part of identifier recognition, determines if it is the same as the keyword after recognition.

```
1  if (strcmp(idStr.c_str(), "return") == 0)
2  {
3      return tok_return;
4  }
5  if (strcmp(idStr.c_str(), "var") == 0)
6  {
7      return tok_var;
8  }
9  if (strcmp(idStr.c_str(), "def") == 0)
10 {
11     return tok_def;
12 }
13
```

2. Identifier Identification:

- (a) Determines if lastChar is a `_` or a letter, if it is it goes to identifier judgement. Define string `idStr`, loop to collect token characters and judge whether it is legal and whether it is a keyword, if it meets the condition then assign the string `idStr` to `identifierStr` to store for subsequent output.

```

1  if (isalpha(lastChar) || lastChar == '_')
2  {
3      std::string idStr;
4      do {
5          idStr += lastChar;
6          lastChar = Token(getNextChar());
7      } while (isalnum(lastChar) || lastChar == '_');
8
9      //Legitimacy checking and keyword detection
10
11     identifierStr = idStr;
12     return tok_identifier;
13 }
14

```

- (b) Illegal identifier recognition

Use `find()` to check whether there are continuous `'_'`

```

1  // no continuous '_'
2  if (idStr.find("__") != std::string::npos)
3  {
4      llvm::errs() << "Identifier <" << idStr << "> contains continuous
5      '_'\n";
6      identifierStr = "ERROR_IDENTIFIER";
7      return tok_identifier;
8  }

```

Use `find()` to check whether there's `'_'` at the beginning.

```

1  // no '_' at the beginning
2  if (idStr.find("_") == 0)
3  {
4      llvm::errs() << "Identifier <" << idStr << "> start with '_'\n";
5      identifierStr = "ERROR_IDENTIFIER";
6      return tok_identifier;
7  }
8

```

Use `find_first_of()` and `find_first_not_of()` to check whether there's number exists but not only at the end.

```

1  // digit only at the end
2  if (idStr.find_first_of("0123456789") != std::string::npos)
3  {
4      if (idStr.find_first_not_of("0123456789", idStr.find_first_of(
5      ("0123456789"))) != std::string::npos) {
6          llvm::errs() << "Identifier <" << idStr << "> end with number\n
7      ";
8          identifierStr = "ERROR_IDENTIFIER";
9          return tok_identifier;
10     }
11 }

```

3. Illegal Number Identification: Similar to the illegal identifier detection above

```

1  // no continuous '.'
2  if (numStr.find("..") != std::string::npos)
3  {
4      llvm::errs() << "Invalid number: " << numStr << "\n";
5      identifierStr = "ERROR_NUMBER";
6      return tok_identifier;
7  }
8  // no more than one '.'
9  if (numStr.find_first_of(".") != numStr.find_last_of("."))
10 {
11     llvm::errs() << "Invalid number: " << numStr << "\n";
12     identifierStr = "ERROR_NUMBER";
13     return tok_identifier;
14 }
15 // no '.' at the beginning
16 if (numStr.find(".") == 0)
17 {
18     llvm::errs() << "Invalid number: " << numStr << "\n";
19     identifierStr = "ERROR_NUMBER";
20     return tok_identifier;
21 }
22

```

2.2 Lexical Analysis Verification Procedure Implementation

Define a vector of type string and traverse the input file with Lexer to get the corresponding tok return value. If it is a keyword, character or EOF, tokenName is the corresponding value; if it is an identifier, use **getId()** to get the name of the identifier; if it is a number, use **getValue()** to get the corresponding value, and convert the floating-point number to a string type to remove the trailing redundant 0.

It is worth noting that the lexer pointer is already pointing to the first token before traversal, and it is necessary for the token to first use **getCurToken()** to get the token it is currently pointing to.

```

1  std::vector<std::string> tokens;
2  auto token = lexer.getCurToken();
3  while (true)
4  {
5      std::string tokenName;
6      switch (token)
7      {
8          case Token::tok_eof:
9              tokenName = "EOF";
10             break;
11          case Token::tok_return:
12              tokenName = "return";
13              break;
14          case Token::tok_def:
15              tokenName = "def";
16              break;
17          case Token::tok_var:
18              tokenName = "var";
19              break;
20          case Token::tok_identifier:

```

```

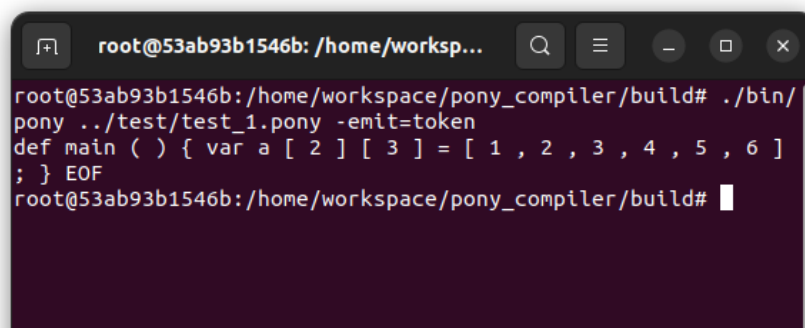
21     tokenName = std::string(lexer.getId());
22     break;
23 case Token::tok_number:
24     tokenName = std::to_string(lexer.getValue());
25     tokenName = tokenName.substr(0, tokenName.find_last_not_of('0') + 1);
26     if (tokenName.back() == '.')
27     {
28         tokenName=tokenName.substr(0, tokenName.size()-1);
29     }
30     break;
31 default:
32     tokenName = std::string(1, token);
33     break;
34 }
35 tokens.push_back(tokenName);
36 //break when eof
37 if (token == Token::tok_eof)
38 {
39     break;
40 }
41 token=lexer.getNextToken();
42 }
43 //print the token by order
44 for (auto &token : tokens) {
45     llvm::outs() << token << " ";
46 }
47 llvm::outs() << "\n";

```

3 Results

3.1 Correct input

test1.pony



```

root@53ab93b1546b: /home/worksp...
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_1.pony -emit=token
def main ( ) { var a [ 2 ] [ 3 ] = [ 1 , 2 , 3 , 4 , 5 , 6 ] ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#

```

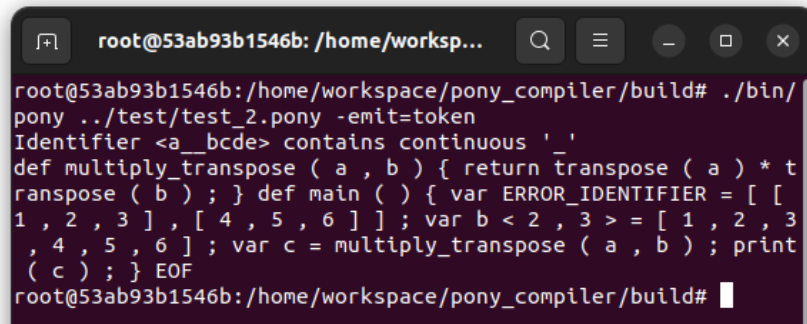
Figure 1: test1

3.2 Error identifier

test2.pony: Identifier contains continous '_'

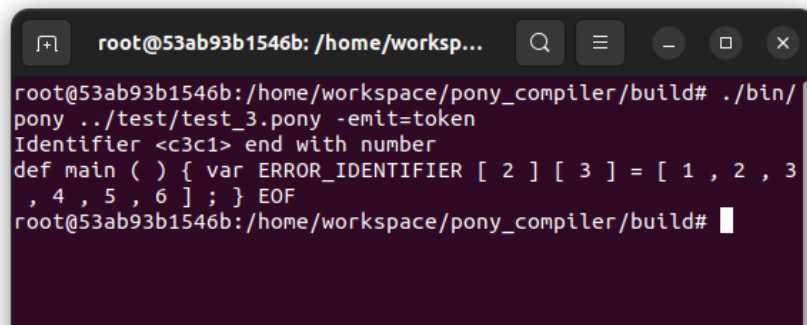
test3.pony: Identifier ends with number.

test4.pony: Identifier begins with '_'



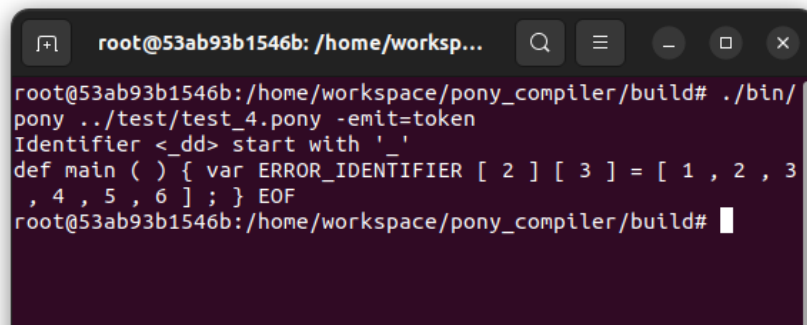
```
root@53ab93b1546b: /home/worksp...
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_2.pony -emit=token
Identifier <a_bcde> contains continuous '_'
def multiply_transpose ( a , b ) { return transpose ( a ) * transpose ( b ) ; } def main ( ) { var ERROR_IDENTIFIER = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ] ; var b < 2 , 3 > = [ 1 , 2 , 3 , 4 , 5 , 6 ] ; var c = multiply_transpose ( a , b ) ; print ( c ) ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 2: test2



```
root@53ab93b1546b: /home/worksp...
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_3.pony -emit=token
Identifier <c3c1> end with number
def main ( ) { var ERROR_IDENTIFIER [ 2 ] [ 3 ] = [ 1 , 2 , 3 , 4 , 5 , 6 ] ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 3: test3

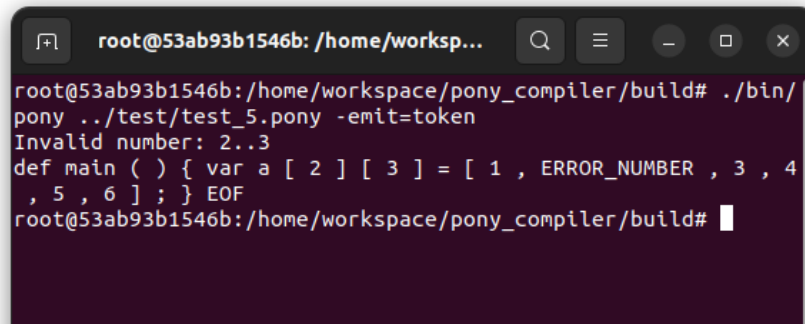


```
root@53ab93b1546b: /home/worksp...
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_4.pony -emit=token
Identifier <_dd> start with '_'
def main ( ) { var ERROR_IDENTIFIER [ 2 ] [ 3 ] = [ 1 , 2 , 3 , 4 , 5 , 6 ] ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 4: test4

3.3 Error number

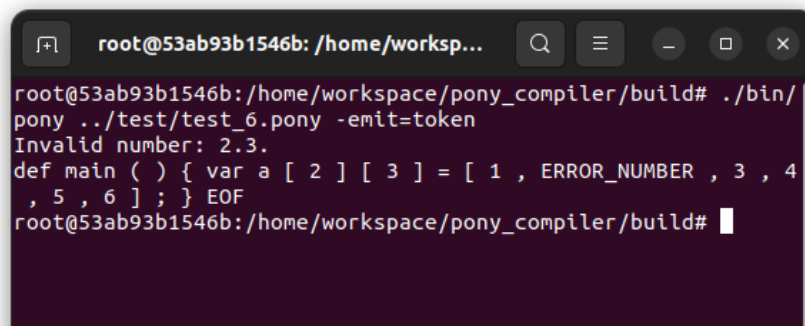
test5.pony: Number contains continuous '.'

A terminal window with a dark purple background. The title bar shows 'root@53ab93b1546b: /home/worksp...'. The command prompt is 'root@53ab93b1546b:/home/workspace/pony_compiler/build#'. The user enters './bin/pony ../test/test_5.pony -emit=token'. The output is 'Invalid number: 2..3' followed by a multi-line error message: 'def main () { var a [2] [3] = [1 , ERROR_NUMBER , 3 , 4 , 5 , 6] ; } EOF'.

```
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_5.pony -emit=token
Invalid number: 2..3
def main ( ) { var a [ 2 ] [ 3 ] = [ 1 , ERROR_NUMBER , 3 , 4 , 5 , 6 ] ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 5: test5

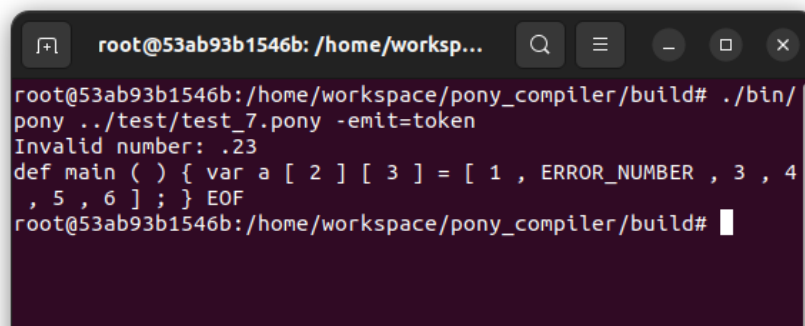
test6.pony: Number contains multiple '.'

A terminal window with a dark purple background. The title bar shows 'root@53ab93b1546b: /home/worksp...'. The command prompt is 'root@53ab93b1546b:/home/workspace/pony_compiler/build#'. The user enters './bin/pony ../test/test_6.pony -emit=token'. The output is 'Invalid number: 2.3.' followed by a multi-line error message: 'def main () { var a [2] [3] = [1 , ERROR_NUMBER , 3 , 4 , 5 , 6] ; } EOF'.

```
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_6.pony -emit=token
Invalid number: 2.3.
def main ( ) { var a [ 2 ] [ 3 ] = [ 1 , ERROR_NUMBER , 3 , 4 , 5 , 6 ] ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 6: test6

test7.pony: Number begins with '.'

A terminal window with a dark purple background. The title bar shows 'root@53ab93b1546b: /home/worksp...'. The command prompt is 'root@53ab93b1546b:/home/workspace/pony_compiler/build#'. The user enters './bin/pony ../test/test_7.pony -emit=token'. The output is 'Invalid number: .23' followed by a multi-line error message: 'def main () { var a [2] [3] = [1 , ERROR_NUMBER , 3 , 4 , 5 , 6] ; } EOF'.

```
root@53ab93b1546b:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_7.pony -emit=token
Invalid number: .23
def main ( ) { var a [ 2 ] [ 3 ] = [ 1 , ERROR_NUMBER , 3 , 4 , 5 , 6 ] ; } EOF
root@53ab93b1546b:/home/workspace/pony_compiler/build#
```

Figure 7: test7