# Test2_redo
# 8 fold assignment

a. Rules for recognizing lexemes and token codes:

• Identifiers:
   - Token code: 100
   - Rules: An identifier must begin with a letter (a-z, A-Z) or an underscore (_) followed by any number of letters, digits (0-9) or underscores.

• Keywords:
   - Token code: 200
   - Rules: Keywords are predefined names that have special meaning in a programming language. The keywords in this language are "while", "for", "do", "if", "int", "short", "long".

• Operators:
   - Token code: 300
   - Rules: Operators are symbols or words that represent an operation to be performed. The operators used in this language are +, -, *, /, %, and =.

• Separators:
   - Token code: 400
   - Rules: Separators are symbols that are used to separate parts of a statement. The separators used in this language are ( ), { }, [ ], ;, and ,.

• Numbers:
   - Token code: 500
   - Rules: Numbers are used to represent a numerical value. The numbers used in this language are integers and floating-point numbers.

• Strings:
   - Token code: 600
   - Rules: Strings are used to represent a sequence of characters. The strings used in this language are enclosed in double-quotes (").

b. Production rules for implementing mathematical syntax:

• Variable declaration:
   - Rule: <variable> ::= <identifier>

• Arithmetic operations:
   - Rule: <expression> ::= <term> | <expression> <operator> <term>
   - Rule: <term> ::= <factor> | <term> <operator> <factor>
   - Rule: <factor> ::= <number> | <identifier> | (<expression>)

• Selection statements:
   - Rule: <selection> ::= if (<condition>) <statement>
   - Rule: <condition> ::= <expression> <relational operator> <expression>
   - Rule: <relational operator> ::= < | > | <= | >= | == | !=

• Loops:
   - Rule: <loop> ::= while (<condition>) <statement>

• Statements:
   - Rule: <statement> ::= <assignment> | <selection> | <loop>
   - Rule: <assignment> ::= <identifier> = <expression>

c. The rules of the language conform to the standard of an LL grammar.

d. The grammar is not ambiguous.

e. A program to process lexemes and produce tokens:

```
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;

// token codes
const int IDENTIFIER = 100;
const int KEYWORD = 200;
const int OPERATOR = 300;
const int SEPARATOR = 400;
const int NUMBER = 500;
const int STRING = 600;

// check if character is letter
bool isLetter(char c){
   return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
}

// check if character is digit
bool isDigit(char c){
   return c >= '0' && c <= '9';
}

// check if character is operator
bool isOperator(char c){
   return c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '=';
}

// check if character is separator
bool isSeparator(char c){
   return c == '(' || c == ')' || c == '{' || c == '}' || c == '[' || c == ']' || c == ';' || c == ',';
}

// check if string is keyword
bool isKeyword(string s){
   return s == "while" || s == "for" || s == "do" || s == "if" || s == "int" || s == "short" || s == "long";
}

int main(){
   // input string
   string str;
   cout << "Enter string: ";
   getline(cin, str);
   // tokens vector
   vector<pair<int, string>> tokens;
   // string buffer
   string buf = "";
   // loop through characters
   for(int i=0; i<str.length(); i++){
      char c = str[i];
      // if character is letter
      if(isLetter(c)){
         buf += c;
         // if next character is not letter or digit
         if(!isLetter(str[i+1]) && !isDigit(str[i+1])){
            // check if string is keyword
```

```
                if(isKeyword(buf)){
                    tokens.push_back({KEYWORD, buf});
                }
                else{
                    tokens.push_back({IDENTIFIER, buf});
                }
                buf = "";
            }
        }
        // if character is digit
        else if(isDigit(c)){
            buf += c;
            // if next character is not digit
            if(!isDigit(str[i+1])){
                tokens.push_back({NUMBER, buf});
                buf = "";
            }
        }
        // if character is operator
        else if(isOperator(c)){
            tokens.push_back({OPERATOR, string(1, c)});
        }
        // if character is separator
        else if(isSeparator(c)){
            tokens.push_back({SEPARATOR, string(1, c)});
        }
        // if character is double quote
        else if(c == '"'){
            // loop until closing double quote
            i++;
            while(str[i] != '"'){
                buf += str[i];
                i++;
            }
            tokens.push_back({STRING, buf});
            buf = "";
        }
    }
    // print tokens
    cout << "Tokens: " << endl;
    for(auto t : tokens){
        cout << t.first << ": " << t.second << endl;
    }
    return 0;
}
```

f. A program or an extension to the above program that determines if the tokens conform to the correct syntax:

```
#include <iostream>
  #include <string>
  #include <vector>
  using namespace std;
// token codes
const int IDENTIFIER = 100;
const int KEYWORD = 200;
const int OPERATOR = 300;
const int SEPARATOR = 400;
const int NUMBER = 500;
const int STRING = 600;
// check if character is letter
bool isLetter(char c){
  return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
}
// check if character is digit
bool isDigit(char c){
  return c >= '0' && c <= '9';
}
// check if character is operator
bool isOperator(char c){
  return c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '=';
}
// check if character is separator
bool isSeparator(char c){
  return c == '(' || c == ')' || c == '{' || c == '}' || c == '[' || c == ']' || c == ';' || c == ',';
}
// check if string is keyword
bool isKeyword(string s){
  return s == "while" || s == "for" || s == "do" || s == "if" || s == "int" || s == "short" || s == "long";
}
int main(){
  // input string
  string str;
  cout << "Enter string: ";
  getline(cin, str);
  // tokens vector
  vector<pair<int, string>> tokens;
  // string buffer
  string buf = "";
  // loop through characters
  for(int i=0; i<str.length(); i++){
    char c = str[i];
    // if character is letter
    if(isLetter(c)){
      buf += c;
      // if next character is not letter or digit
      if(!isLetter(str[i+1]) && !isDigit(str[i+1])){
        // check if string is keyword
        if(isKeyword(buf)){
          tokens.push_back({KEYWORD, buf});
        }
        else{
          tokens.push_back({IDENTIFIER, buf});
        }
        buf = "";
      }
    }
    // if character is digit
    else if(isDigit(c)){
      buf += c;
      // if next character is not digit
      if(!isDigit(str[i+1])){
        tokens.push_back({NUMBER, buf});
        buf = "";
      }
    }
    // if character is operator
    else if(isOperator(c)){
      tokens.push_back({OPERATOR, string(1, c)});
                    }
                    // if character is separator
                    else if(isSeparator(c)){
        tokens.push_back({SEPARATOR, string(1, c)});
                      }
                      // if character is double quote
                      else if(c == '"'){
          // loop until closing double quote
          i++;
          while(str[i] != '"'){
            buf += str[i];
            i++;
          }
          tokens.push_back({STRING, buf});
          buf = "";
        }
      }
      // syntax checking
      bool valid = true;
      for(int i=0; i<tokens.size(); i++){
        int code = tokens[i].first;
        string str = tokens[i].second;
        // check for variable declaration
        if(code == IDENTIFIER && i < tokens.size()-1 && tokens[i+1].first == OPERATOR &&
tokens[i+1].second == "="){
          // valid variable declaration
        }
        // check for arithmetic operations
          else if(code == OPERATOR && str == "+" || str == "-" || str == "*" || str == "/" || str == "%")
{
            if(i < tokens.size()-2 && tokens[i+1].first == NUMBER && tokens[i+2].first == NUMBER){
              // valid arithmetic operation
            }
            else{
              valid = false;
            }
          }
        // check for selection statements
        else if(code == KEYWORD && str == "if"){
          if(i < tokens.size()-5 && tokens[i+1].first == SEPARATOR && tokens[i+1].second == "(" &&
tokens[i+2].first == NUMBER && tokens[i+3].first == OPERATOR && tokens[i+4].first ==
NUMBER && tokens[i+5].first == SEPARATOR && tokens[i+5].second == ")"){
            // valid selection statement
          }
          else{
            valid = false;
          }
        }
        // check for loops
        else if(code == KEYWORD && str == "while"){
          if(i < tokens.size()-5 && tokens[i+1].first == SEPARATOR && tokens[i+1].second == "(" &&
tokens[i+2].first == NUMBER && tokens[i+3].first == OPERATOR && tokens[i+4].first ==
NUMBER && tokens[i+5].first == SEPARATOR && tokens[i+5].second == ")"){
            // valid loop
          }
          else{
            valid = false;
          }
        }
      }
      // print result
      if(valid){
        cout << "Syntax is valid" << endl;
      }
      else{
        cout << "Syntax is invalid" << endl;
      }
      return 0;
}
```

g. Test files:

Test File 1 (with lexical errors):

```
whille 2+3
```

Test File 2 (with syntax errors):

```
int x = 2 + ;
```

Test File 3 (no errors):

```
int x = 2 + 3;
```

Test File 4 (no errors):

```
while (x > 5){
  x = x + 1;
}
```