CSC 306 LEXICAL ANALYSIS PROJECT
SPRING 2017

PROF. GODFREY C. MUGANDA
DEPARTMENT OF COMPUTER SCIENCE

In this assignment, we take the first step in writing a compiler for the NINC programming language, and construct a lexical analyzer for NINC. The lexical analyzer will take a NINC source file as input, and output a stream of tokens making up the NINC program in the file. In addition, the lexical analyzer strips out comments, making them invisible to subsequent phases of the compiler.

Comments start with // and end at the end of the line or the end of file. There is also a multi-line line form of commenting that starts with the character combination /* and ends with the character combination */. Multi-line comments may not be nested.

Identifier tokens consist of a sequence of alphanumeric characters and underscores. Identifiers may not start with a digit and are case sensitive.

Number tokens are sequences of digits.

Strings tokens consist of a sequence of characters delimited by double quotes.

Each token is formed by the longest sequence of characters that obey the rules for the formation of that token.

Here is a table of NINC reserved words:

boolean	**break**	**case**	**cin**
continue	**cout**	**endl**	**else**
exit	**false**	**if**	**int**
main	**switch**	**true**	**void**
while			

The rest of the tokens are given in Table 1.

For this project, your project will take 2 file names on the command line: the first file will be a NINC source file, and the second file will be a listing file. For this assignment, the listing file will merely be a copy of the source file: subsequent assignments will use the listing file to indicate positions of syntax errors within the source.

Category of Token	tokens
Punctuation symbols	, ; : ( ) { }
Relational operators	== != < <= > >=
Arithmetic operators	- + * / %
Boolean operators	! || &&
Input and Output operators	<< >>
Assignment operator	=

Table 1. Symbols and operators used in NINC

The job of the lexical analyzer being to break source code into a sequence of tokens, the output of your program will consist of token-lexeme pairs (the token is really an internal representation, while the lexeme is the string that actually appears in the source program.) For example, when the source file is

```
void main( )
{
  cout << "Hello World";
}
```

is fed to the lexical analyzer as input, the output will be

```
t_void  void
t_main  main
t_lparen  (
t_rparen  )
t_lbrace  {
t_cout  cout
t_insert  <<
t_string  Hello World
t_semicolon  ;
t_rbrace  }
```

The internal representation of a token is an enumeration type, which makes it an integer. We have assigned readable names to each enumeration type of value: these names start with the prefix `t_`.

1. Additional Requirements

Your lexical analyzer must meet the following additional requirements:

(1) It should throw a string exception

```
Unexpected end of input while scanning a string.
```

or

```
Unexpected end of input while scanning a comment.
```

if the input file ends with an un-closed string or multi-line comment. This happens if the programmer forgets the closing double quote " on a string or the closing */ characters on a multi-line comment.

You may assume that no string will contain an embedded double quote ", so the first double quote after the opening double quote ends the string.

(2) No string may contain a line break, so the program should throw a string exception

```
Found line break while scanning a string.
```

if a line break is encountered while scanning a string.

(3) It should return `Token::EOS` at the end of the source file. Note that may occur at the end of a single line comment.

(4) It a character that does not start any token is encountered, the scanner should skip it and return a `Token::ERROR`.

2. Getting Started

Organize your lexical analyzer in three files. Two of the files, `lexer.h` and `lex.cpp` specify the lexical analyzer and provide the implementation, respectively, and the third file, `main.cpp`, is the driver file used for testing.

The job of a scanner is to transform a sequence of characters, read from an input file, into a sequence of tokens that have meaning for the language. The different tokens for NINC are given in the following enumeration type:

```
enum class Token
{
   // reserved key words
   BOOL, BREAK, CASE, CIN, CONTINUE, COUT, ENDL, ELSE,
   EXIT, FALSE, IF, INT, MAIN, SWITCH, TRUE, VOID, WHILE,
   // punctuation
   COMMA, SEMICOLON, COLON, LPAREN, RPAREN, LBRACE, RBRACE,
   // arithmetic operators
   MINUS, PLUS, MULT, DIV, MOD,
   // relational operators
   EQ, NE, LT, LE, GT, GE,
   // boolean operators
   NOT, OR, AND,
   // input and output
   EXTRACT, INSERT,
   // Assignment
   ASSIGN,
   // ID, NUMBER, and STRING
   ID, NUM, STR,
```

```
   // Unknown token, end of input stream
   ERROR, EOS
};
```

Internally, these are represented by non-negative integers, according to the order given. Thus `Token::BOOL` is 0, `Token::BREAK` is 1, and so on. You can convert between the `Token` type and `int` by using static casts.

Corresponding to `Token`, we define a vector of strings that will allow us to perform formatted output of `Token` types:

```
// vector maps TokenType to string
vector<string> tokenToStringVector
{
    // keywords
    "t_bool", "t_break", "t_case", "t_cin", "t_continue",
    "t_cout", "t_endl", "t_else", "t_exit", "t_false",
    "t_if", "t_int", "t_main", "t_switch", "t_true",
    "t_void", "t_while",
    // punctuation
    "t_comma", "t_semicolon", "t_colon", "t_lparen",
    "t_rparen", "t_lbrace", "t_rbrace",
     // arithmetic
    "t_minus", "t_plus", "t_mult", "t_div", "t_mod",
    // relational
    "t_eq", "t_ne", "t_lt", "t_le", "t_gt", "t_ge",
    // boolean
    "t_not", "t_or", "t_and",
    // input output
    "t_extract", "t_insert",
    // assign
    "t_assign",
    // others
    "t_id", "t_number", "t_string", "t_error", "t_eof"
};
```

Keywords have the same syntax as identifiers, so we maintain a list of NINC keywords, to help us recognize keywords and distinguish them from identifiers. The vector of keywords will be used to build a map that associates a keyword with its corresponding token:

```
// list of keywords
vector<string> keywords
{
   "bool", "break", "case", "cin", "continue", "cout", "endl",
   "else", "exit", "false", "if", "int", "main", "switch", "true",
   "void", "while"
};

// maps keyword strings identifier tokens, initialized in constructor
```

```
// from the keywords vector
map<string, Token> keywordTokenMap;
```

The most important members of the lexical analyzer class `Lexer` are

(1) A constructor `Lexer(istream &source, ostream& listing)` that takes an input stream containing NINC source code, and an output stream to which to write the a listing of the source as it is being processed.
(2) A member function `Token getToken()`: each call to this function returns the next token available from the source stream.
(3) A member function `string getLexeme()`: each call to this function returns the lexeme corresponding to token just returned by `getToken()`. This is needed because some tokens such as `Token::NUM` and `Token::STR` have different lexemes.

3. The Lexer Class Header File

The complete code for the `Lexer` class is given here. This is complete and can be used as is.

```
#pragma once
/*
* File:   Lexer.h
* Author: gcm
*/

#ifndef LEXER_H
#define LEXER_H
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <vector>
#include <map>
using namespace std;
enum class Token
{
   // reserved key words
   BOOL, BREAK, CASE, CIN, CONTINUE, COUT, ENDL, ELSE,
   EXIT, FALSE, IF, INT, MAIN, SWITCH, TRUE, VOID, WHILE,
   // punctuation
   COMMA, SEMICOLON, COLON, LPAREN, RPAREN, LBRACE, RBRACE,
   // arithmetic operators
   MINUS, PLUS, MULT, DIV, MOD,
   // relational operators
   EQ, NE, LT, LE, GT, GE,
   // boolean operators
   NOT, OR, AND,
   // input and output
```

```
   EXTRACT, INSERT,
   // Assignment
   ASSIGN,
   // ID, NUMBER, and STRING
   ID, NUM, STR,
   // Unknown token, end of input stream
   ERROR, EOS
};


class Lexer
{
public:

   Lexer(istream& source, ostream &listing);
   // Determine the next token in the source,
   // set lexeme, and return the token

   Token getToken();
   // Return the lexeme corresponding to the
   //last token returned by getToken()

   string getLexeme() { return lexeme; }
   // Return printable string form of a token

   string stringify(Token t)
   {
     if (t == Token::ERROR)
        return "Error!";
     else
        return tokenToStringVector[static_cast<int>(t)];
   }

private:

   // Check if  "str" is a keyword, and if so,
   // set "t" to the token
   // corresponding to the keyword.
   bool isKeyWord(string str, Token & t);

   // Get the next char from source, echo to listing,
   // and return the char
   char getChar()
   {
      char ch = source.get();
      if (ch != EOF) listing.put(ch);
      return ch;
   }
```

```
// source and listing files
istream& source;
ostream& listing;

// lexeme for the token that was last returned by getToken
string lexeme;

// vector maps TokenType to string
vector<string> tokenToStringVector
{
   // keywords
   "t_bool", "t_break", "t_case", "t_cin", "t_continue",
   "t_cout", "t_endl", "t_else", "t_exit", "t_false",
   "t_if", "t_int",  "t_main", "t_switch", "t_true",
   "t_void", "t_while",
   // punctuation
   "t_comma", "t_semicolon", "t_colon", "t_lparen",
   "t_rparen", "t_lbrace", "t_rbrace",
   // arithmetic
   "t_minus", "t_plus", "t_mult", "t_div", "t_mod",
    // relational
   "t_eq", "t_ne", "t_lt", "t_le", "t_gt", "t_ge",
   // boolean
   "t_not", "t_or", "t_and",
   // input output
   "t_extract", "t_insert",
   // assign
   "t_assign",
   // others
   "t_id", "t_number", "t_string", "t_error", "t_eof"
};

// list of keywords
vector<string> keywords
{
   "bool", "break", "case", "cin", "continue", "cout",
   "endl",  "else", "exit", "false", "if", "int",
   "main", "switch", "true", "void", "while"
};

// Maps keyword strings identifier tokens, initialized
// in constructor  from the keywords vector
map<string, Token> keywordTokenMap;

// single char token map: used to simplify mapping of
// lexemes to tokens
map<string, Token> singleCharTokenMap
```

```
   {
      {",", Token::COMMA},
      {";", Token::SEMICOLON},
      {":", Token::COLON},
      {"(", Token::LPAREN},
      {")", Token::RPAREN},
      {"{", Token::LBRACE},
      {"}", Token::RBRACE},
      {"-", Token::MINUS},
      {"+", Token::PLUS},
      {"*", Token::MULT},
      {"%", Token::MOD}
   };
};
#endif /* LEXER_H */
```

4. The Lexer Class Implementation File

Here is a shell of what needs to be implemented. You just need to write the `getToken()` function.

```
#include "Lexer.h"
#include <algorithm>
#include <cctype>

// Initializes the source and listing streams via
// constructor initialization syntax
// and sets up the lexemeTokenMap to map reserved keyword
// to their corresponding tokens.
Lexer::Lexer(istream &source1, ostream& listing1)
     :source(source1), listing(listing1)
{
   // initialize the lexemeTokenMap to map keywords to their tokens
   for (int k = 0; k < keywords.size(); k++)
   {
      keywordTokenMap[keywords[k]] = static_cast<Token>(k);
   }
}

// Determine the next token, set lexeme, and return the token
// PRE: ch is blank initially, or holds a character at or before
// the first character of the token to be returned
// POST: ch holds the first character after the returned lexeme
Token Lexer::getToken()
{



}
```

```
// Checks the vector of keywords to see if str is a keyword,
// if so, uses the lexemeTokenMap to map the keyword to a
// token and sets t to the value of that token and returns
// true. Otherwise it returns false.
bool Lexer::isKeyWord(string str, Token& t)
{
   vector<string>::iterator
       it = std::find(keywords.begin(), keywords.end(), str);

   if (it == keywords.end()) return false;
   else
   {
      t = keywordTokenMap[str];
      return true;
   }
}
```

5. The Driver File

This will be used to test the lexical analyzer. This is complete and can be used as is.

```
#include <cstdlib>
#include "Lexer.h"
#include <sstream>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

void processCmdLine(int, char **, ifstream&, ofstream&);

int main(int argc, char** argv)
{
    /*
    istringstream is
    (
            "bool break case cin continue cout endl else exit false "
            "if int main switch true void while "
            ", ; : ( ) { } - + * / %  == != < <= > >= "
            "! || && >> << = id 123 "
    );
     */

    //istringstream is("bool break /*This is a comment */ 12 id");

    //Lexer lex(is, cout);
```

```
    ifstream sourceFile;
    ofstream listingFile;

    processCmdLine(argc, argv, sourceFile, listingFile);
    Lexer lex(sourceFile, listingFile);

    try
    {
       string lexeme;
       Token t = lex.getToken();

       while (t != Token::EOS)
       {
          lexeme = lex.getLexeme();
          cout << " " << lex.stringify(t) << "  " << lexeme << endl;
          t = lex.getToken();
       }
       system("pause");
       return 0;
    }
    catch (char const * exception)
    {
        cout << exception << endl;
    }
}


// process the command line args and set the input and listing streams

void processCmdLine(int argc, char **argv, i
                    fstream &sourceFile, ofstream &listingFile)
{
    if (argc != 3)
    {
        cout << "You need to provide a source and listing file on the command line"
             << endl;
        cout << argv[0] << " source_file listing_file" << endl;
        exit(1);
    }
    // Open the source file
    sourceFile.open(argv[1]);
    if (!sourceFile)
    {
        cout << "Cannot open the file " << argv[1] << endl;
        exit(2);
    }
    // Open the listing file
    listingFile.open(argv[2]);
```

```
    if (!listingFile)
    {
        cout << "Cannot open the file " << argv[1] << endl;
        exit(3);
    }
}


// process the command line args and set the input
// and listing streams
void  processCmdLine(int argc, char **argv,
                     ifstream &sourceFile,
                     ofstream &listingFile)
{
   if (argc != 3)
   {
      cout << "You need to provide a source and "
           << "listing file on the command line"
           << endl;
      cout << argv[0] << " source_file listing_file" << endl;
      exit(1);
   }
   // Open the source file
   sourceFile.open(argv[1]);
   if (!sourceFile)
   {
      cout << "Cannot open the file " << argv[1] << endl;
      exit(2);
   }
   // Open the listing file
   listingFile.open(argv[2]);
   if (!listingFile)
   {
      cout << "Cannot open the file " << argv[1] << endl;
      exit(3);
   }
}
```