

**МIНIСТЕРСТВО  ОСВIТИ І НАУКИ  УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

“**КИЇВСЬКИЙ  ПОЛІТЕХНІЧНИЙ  ІНСТИТУТ**

**ІМЕНІ ІГОРЯ СІКОРСЬКОГО”**

**Факультет прикладної математики**

**Кафедра програмного забезпечення комп’ютерних систем**

**Лабораторна робота № 9**

**з дисципліни “ Основи програмування ”**

**тема “ Файлові потоки ”**

|  |  |  |
| --- | --- | --- |
| **Виконала**  **студентка I курсу**  **групи КП-83**  **Снітко Маріанна Дмитрівна**  **(прізвище, ім’я, по батькові)**  **варіант № 22** |  | **Перевірив**  “**\_\_\_\_” “\_\_\_\_\_\_\_\_\_\_\_\_” 20\_\_\_ р.**  **викладач**  **Гадиняк Руслан Анатолійович**  **(прізвище, ім’я, по батькові)** |

**Київ 2018**

**Мета роботи**

Навчитися працювати із текстовими файловими потоками.

**Постановка завдання**

Скопіювати у директорію даного завдання код рішення із попередньої лабораторної роботи та продовжити роботу над копією.

1. Винести вхідний рядок із кодом (за варіатном) у файл input.txt, який розмістити у директорії завдання.
2. Додати обробку аргументів командного рядка для підтримки таких опцій роботи програми:
   1. inputFileName.txt - перший вільний аргумент командного рядка, задає назву вхідного файлу (по аналогії з вхідним файлом в gcc).
   2. -o outputFileName.txt - опція із аргументом, що задає назву вихідного файлу.
   3. -l - (маленька літера L) булева опція без аргумента, її наявність включає режим виводу списку токенів.
   4. (UPD) Наявність будь-яких інших опцій є помилкою, що завершує роботу програми з попереднім виводом першої з цих опцій у консоль (наприклад, error: unrecognized command line option ‘-K’).
   5. Якщо жодних опцій не задано:
      * програма має працювати в стандартному режимі
      * назва вхідного файлу за замовчуванням буде input.txt
      * вивід результату у стандартний файловий потік (консоль).
3. Зчитати весь текст із вхідного файлового потоку, виконати його лексичний аналіз та отримати список токенів (як у попередній лаборатоній роботі). Результат (залежить від режиму роботи) вивести у файловий потік виводу.
4. Покращення лексичного аналізу:
   1. Додати новий тип токена (TOKEN\_WHITESPACE) і при аналізі вхідного рядка зберігати послідовності пустих символів у таблиці whitespaces.
   2. Додати новий тип структур (struct TextPosition { int row, column; };) для зберігання позиції у тексті. Додати у структуру токена поле типу struct TextPosition і при аналізі зберігати в ньому позицію вхідного тексту, де починається лексема даного токена.

Режими роботи:

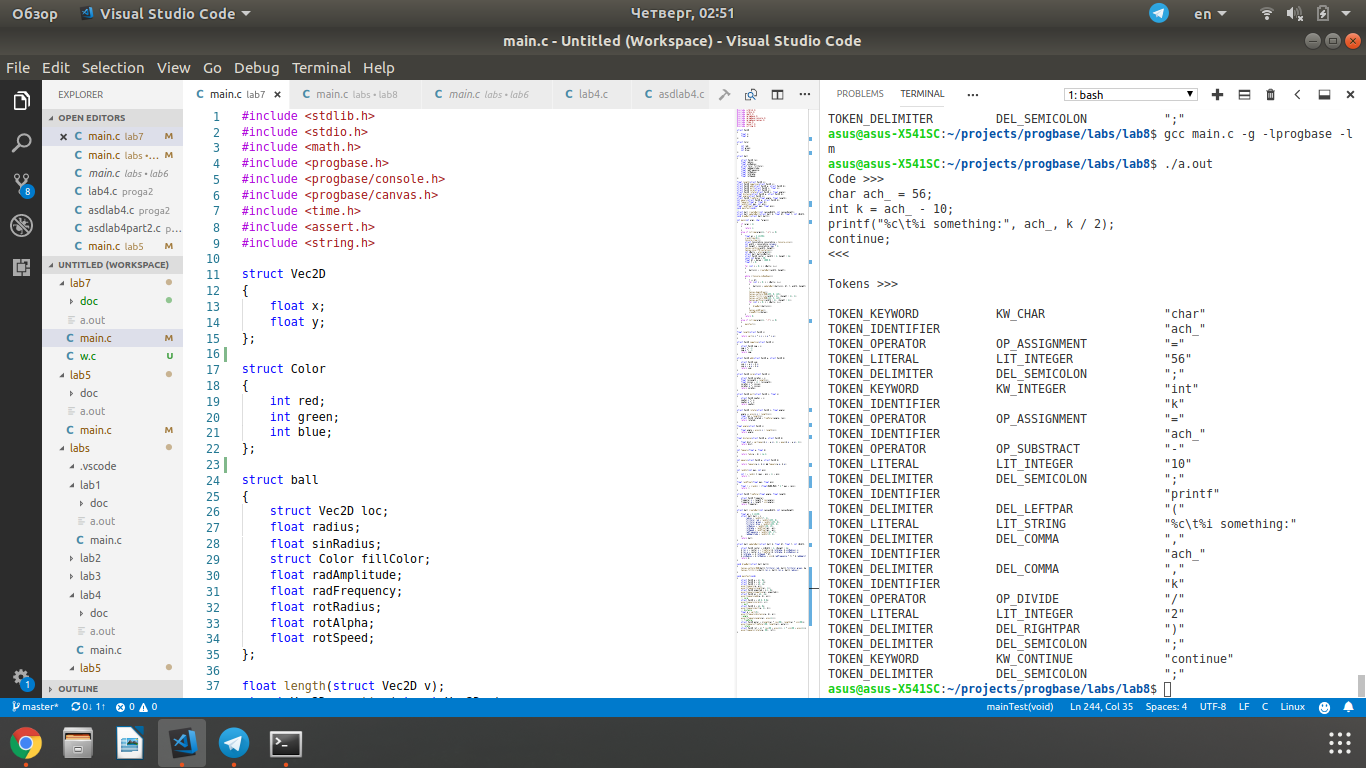
1. Стандартний режим роботи - отримати список токенів і вивести як результат всі лексеми у консоль так, як вони були розташовані у вхідному тексті, задаючи різний колір тексту для лексем різних типів (підсвітка коду).
   * При виводі рядкових літералів не забувати додавати пари подвійних лапок (").
2. Режим виводу списку токенів - вивести як результат текстове представлення списку токенів у форматі з попередньої лабораторної роботи.
   * Додати стовпці для виводу номера рядка та номера стовпця початку лексеми у вхідному тексті. Новий порядок стовпців: Тип токена, Підтип токена, Номер рядка, Номер стовпця, Лексема.
   * Токени типу TOKEN\_WHITESPACE не виводити.

**Тексти коду програм**

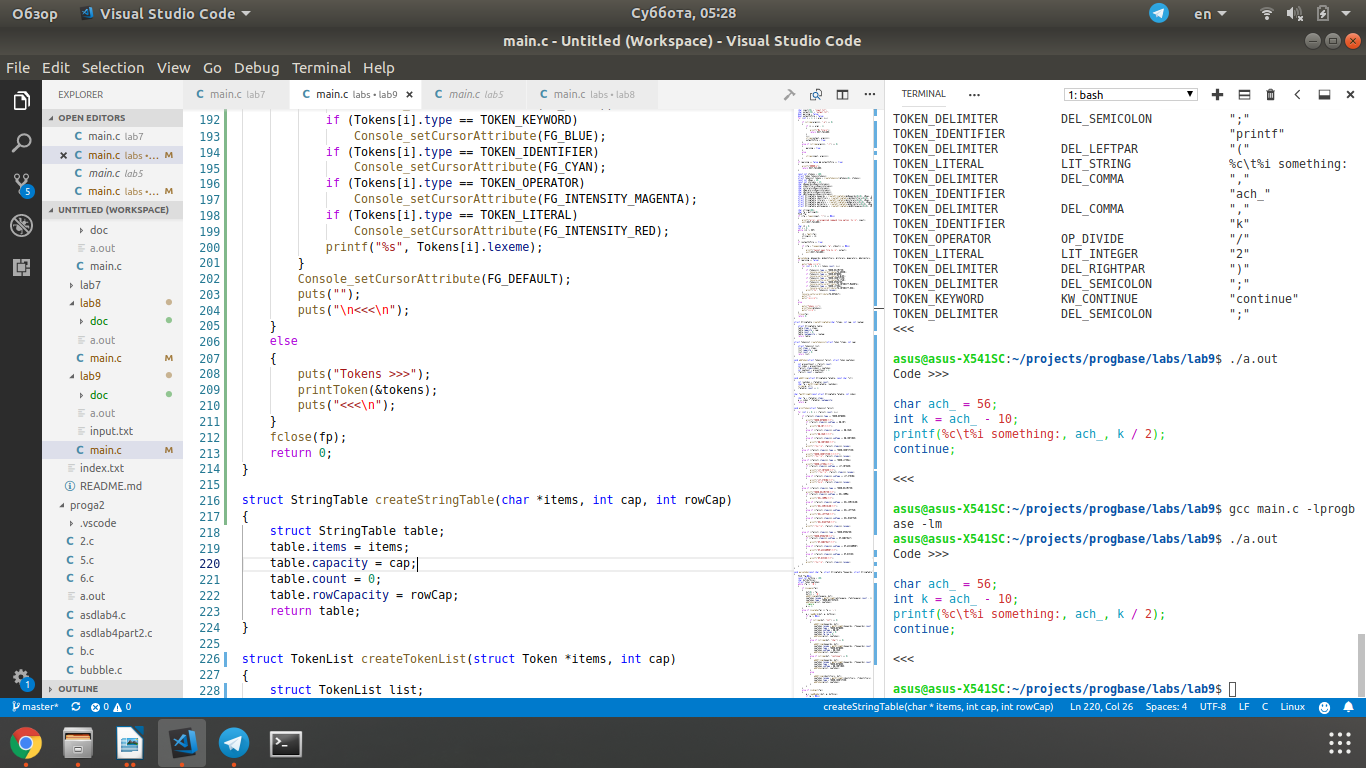
|  |
| --- |
| **main.c** |
| #include <string.h>  #include <ctype.h>  #include <stdlib.h>  #include <stdio.h>  #include <stdbool.h>  #include <progbase.h>  #include <progbase/console.h>  enum TokenType  {  TOKEN\_KEYWORD,  TOKEN\_OPERATOR,  TOKEN\_DELIMITER,  TOKEN\_LITERAL,  TOKEN\_IDENTIFIER,  TOKEN\_WHITESPACE  };  enum TokenKeywords  {  KW\_INTEGER,  KW\_CHAR,  KW\_CONTINUE,  };  enum TokenOperators  {  OP\_ASSIGNMENT,  OP\_SUBSTRACT,  OP\_DIVISION,  };  enum TokenLiterals  {  LIT\_FLOAT,  LIT\_INTEGER,  LIT\_STRING  };  enum TokenDelimeters  {  DEL\_SEMICOLON,  DEL\_COMMA,  DEL\_LEFTPAR,  DEL\_RIGTHPAR,  };  struct TextPosition  {  int row;  int column;  };  struct Token  {  char \*lexeme;  enum TokenType type;  int subType;  struct TextPosition position;  };  struct TokenList  {  struct Token \*items;  size\_t count;  size\_t capacity;  };  struct StringTable  {  char \*items;  size\_t capacity;  size\_t count;  size\_t rowCapacity;  };  void parseCode(const char \*p, bool outputFile, bool tokenList, char \*pOutput);  const char \*readWord(char \*dest, const char \*src, int destLen);  const char \*readNumder(char \*dst, const char \*src, int destLen);  const char \*readString(char \*dst, const char \*src, int destLen);  const char \*reaOpOrDel(char \*dst, const char \*src, int destLen);  const char \*readWhiteSpace(char \*dst, const char \*src, int destLen);  struct StringTable createStringTable(char \*items, int cap, int rowCap);  char \*getStringAt(const struct StringTable \*ptable, int index);  void addString(struct StringTable \*ptable, const char \*str);  char \*findString(const struct StringTable \*ptable, const char \*str);  bool containsString(const struct StringTable \*ptable, const char \*str);  void printTable(const struct StringTable \*ptable);  struct TokenList createTokenList(struct Token \*items, size\_t length);  void addToken(struct TokenList \*ptokens, struct Token token);  void printTokens(const struct TokenList \*ptonkes);  int main(int argc, char \*argv[])  {  char outputFileName[50];  char inputFileName[50] = {"input.txt"};  char incorrectOption[50];  bool isTokensPrintingOut = false;  bool isFileOutputting = false;  bool wasInputName = false;  for (int i = 1; i < argc; i++)  {  char \*p = argv[i];  if (\*p == '-')  {  if (\*(p + 1) == 'l')  {  isTokensPrintingOut = true;  }  else if (\*(p + 1) == 'o' && isFileOutputting == false)  {  if (i == argc - 1)  {  printf("No file name\n");  return 1;  }  strcpy(outputFileName, argv[i + 1]);  isFileOutputting = true;  }  else  {  strcpy(incorrectOption, argv[i]);  printf("error: unrecognized command line option ‘%s’\n", incorrectOption);  return 1;  }  }  if (wasInputName == false && (strcmp(argv[i], "-o") != 0) && (strcmp(argv[i], "-l") != 0) && (strcmp(argv[i], outputFileName) != 0))  {  wasInputName = true;  strcpy(inputFileName, argv[i]);  }  }  FILE \*fp = fopen(inputFileName, "r+");  if (fp == NULL)  {  printf("error: non existant file\n");  exit(EXIT\_FAILURE);  }  char inputString[300];  char ch = 0;  int i = 0;  while (ch != EOF)  {  ch = fgetc(fp);  inputString[i] = ch;  i++;  }  inputString[i - 1] = '\0';  char \*p = inputString;  char \*pOut = outputFileName;  parseCode(p, isFileOutputting, isTokensPrintingOut, pOut);  fclose(fp);  return (0);  }  struct StringTable createStringTable(char \*items, int cap, int rowCap)  {  struct StringTable table;  table.items = items;  table.capacity = cap;  table.count = 0;  table.rowCapacity = rowCap;  return table;  }  char \*getStringAt(const struct StringTable \*ptable, int index)  {  char \*p = (\*ptable).items;  p += index \* (\*ptable).rowCapacity;  return p;  }  void addString(struct StringTable \*ptable, const char \*str)  {  int rowIndex = (\*ptable).count;  int maxRows = (\*ptable).capacity;  if (rowIndex == maxRows)  {  return;  }  if (strlen(str) >= (\*ptable).rowCapacity)  {  return;  }  char \*p = getStringAt(ptable, rowIndex);  strcpy(p, str);  (\*ptable).count += 1;  }  char \*findString(const struct StringTable \*ptable, const char \*str)  {  for (int i = 0; i < (\*ptable).count; i++)  {  char \*p = getStringAt(ptable, i);  if (strcmp(p, str) == 0)  {  return p;  }  }  return NULL;  }  void printTable(const struct StringTable \*ptable)  {  printf("%zu:\n", (\*ptable).count);  for (int i = 0; i < (\*ptable).count; i++)  {  char \*p = getStringAt(ptable, i);  printf("(%d) '%s'\n", i, p);  }  }  bool containsString(const struct StringTable \*ptable, const char \*str)  {  return findString(ptable, str) != NULL;  }  struct TokenList createTokenList(struct Token \*items, size\_t length)  {  struct TokenList list;  list.items = items;  list.capacity = length;  list.count = 0;  return list;  }  void addToken(struct TokenList \*ptokens, struct Token token)  {  int prevCount = (\*ptokens).count;  int nMaxItems = (\*ptokens).capacity;  if (prevCount == nMaxItems)  {  return;  }  int index = prevCount;  (\*ptokens).items[index] = token;  int newCount = prevCount + 1;  (\*ptokens).count = newCount;  }  void printTokens(const struct TokenList \*ptokens)  {  printf("Tokens>>>\n");  for (int i = 0; i < (\*ptokens).count; i++)  {  struct Token t = (\*ptokens).items[i];  if ((\*ptokens).items[i].type == TOKEN\_KEYWORD)          {              printf("TOKEN\_KEYWORD\t\t");              if ((\*ptokens).items[i].subType == KW\_CHAR)              {                  printf("KW\_CHAR\t\t\t");              }              else if ((\*ptokens).items[i].subType == KW\_INTEGER)              {                  printf("KW\_INTEGER\t\t");              }              else if ((\*ptokens).items[i].subType == KW\_CONTINUE)              {                  printf("KW\_CONTINUE\t\t");              }              printf("\"%s\"\t%d\t%d\n", t.lexeme,t.position.row, t.position.column);          }          else if ((\*ptokens).items[i].type == TOKEN\_LITERAL)          {              printf("TOKEN\_LITERAL\t\t");              if ((\*ptokens).items[i].subType == LIT\_INTEGER)              {                  printf("LIT\_INTEGER\t\t");              }              else if ((\*ptokens).items[i].subType == LIT\_STRING)              {                  printf("LIT\_STRING\t\t");              }              printf("\"%s\"\t%d\t%d\n", t.lexeme,t.position.row, t.position.column);          }          else if ((\*ptokens).items[i].type == TOKEN\_IDENTIFIER)          {              printf("TOKEN\_IDENTIFIER\t\t\t\t");              printf("\"%s\"\t%d\t%d\n", t.lexeme,t.position.row, t.position.column);          }          else if ((\*ptokens).items[i].type == TOKEN\_OPERATOR)          {              printf("TOKEN\_OPERATOR\t\t");              if ((\*ptokens).items[i].subType == OP\_ASSIGNMENT)              {                  printf("OP\_ASSIGNMENT\t\t");              }              else if ((\*ptokens).items[i].subType == OP\_SUBSTRACT)              {                  printf("OP\_SUBSTRACT\t\t");              }              else if ((\*ptokens).items[i].subType == OP\_DIVISION)              {                  printf("OP\_DIVISION\t\t");              }              printf("\"%s\"\t%d\t%d\n", t.lexeme,t.position.row, t.position.column);          }          else if ((\*ptokens).items[i].type == TOKEN\_DELIMITER)          {              printf("TOKEN\_DELIMITER\t\t");              if ((\*ptokens).items[i].subType == DEL\_COMMA)              {                  printf("DEL\_COMMA\t\t");              }              else if ((\*ptokens).items[i].subType == DEL\_SEMICOLON)              {                  printf("DEL\_SEMICOLON\t\t");              }              else if ((\*ptokens).items[i].subType == DEL\_LEFTPAR)              {                  printf("DEL\_LEFTPAR\t\t");              }              else if ((\*ptokens).items[i].subType == DEL\_RIGTHPAR)              {                  printf("DEL\_RIGHTPAR\t\t");              }              printf("\"%s\"\t%d\t%d\n", t.lexeme,t.position.row, t.position.column);          }  }  printf("<<<\n");  }  void parseCode(const char \*p, bool outputFile, bool tokenList, char \*pOutput)  {  const int tokensLength = 100;  struct Token aTokens[tokensLength];  struct TokenList tokens = createTokenList(aTokens, tokensLength);  const int nKeywords = 5;  const int nMaxKeywords = 5;  char mKeywords[nKeywords][nMaxKeywords + 1];  struct StringTable keywords = createStringTable(&mKeywords[0][0], nKeywords, nMaxKeywords);  addString(&keywords, "char");  addString(&keywords, "int");  addString(&keywords, "continue");  const int MaxIdentifiers = 100;  const int nMaxIdLength = 100;  char mIdentifiers[MaxIdentifiers][nMaxIdLength + 1];  struct StringTable identifiers = createStringTable(&mIdentifiers[0][0], MaxIdentifiers, nMaxIdLength);  const int nMaxLiterals = 100;  const int nMaxLitLength = 100;  char mLiterals[nMaxLiterals][nMaxLitLength + 1];  struct StringTable literals = createStringTable(&mLiterals[0][0], nMaxLiterals, nMaxLitLength);  const int nMaxDelOrOp = 100;  const int nMaxDelOrOpLength = 2;  char mDelOrOpLength[nMaxLiterals][nMaxLitLength + 1];  struct StringTable DelOrOpLength = createStringTable(&mDelOrOpLength[0][0], nMaxDelOrOp, nMaxDelOrOpLength);  const int nMaxWhiteSpaces = 100;  const int nMaxWhiteSpacesLength = 100;  char mWhiteSpaces[nMaxWhiteSpaces][nMaxWhiteSpacesLength + 1];  struct StringTable whiteSpaces = createStringTable(&mWhiteSpaces[0][0], nMaxWhiteSpaces, nMaxWhiteSpacesLength);  const int bufLen = 100;  char buf[bufLen];  int nRow = 1;  int nColumn = 1;  bool wasError = false;  while (\*p != '\0')  {  if (isspace(\*p))  {  if (\*p == '\n')  {  nRow++;  nColumn = 0;  }  struct Token token1 = {0};  p = readWhiteSpace(buf, p, bufLen);  if (\*p != '\0')  {  if (containsString(&literals, buf))  {  token1.lexeme = findString(&whiteSpaces, buf);  token1.type = TOKEN\_WHITESPACE;  }  else  {  addString(&whiteSpaces, buf);  token1.lexeme = findString(&whiteSpaces, buf);  token1.type = TOKEN\_WHITESPACE;  }  }  token1.position.row = nRow;  token1.position.column = nColumn;  nColumn += strlen(buf);  addToken(&tokens, token1);  }  else if (isalpha(\*p) || \*p == '\_')  {  struct Token token1 = {0};  p = readWord(buf, p, bufLen);  if (\*p != '\0')  {  if (containsString(&keywords, buf))  {  token1.lexeme = findString(&keywords, buf);  token1.type = TOKEN\_KEYWORD;  if (strcmp(buf, "int") == 0)  {  token1.subType = KW\_INTEGER;  }  if (strcmp(buf, "char") == 0)  {  token1.subType = KW\_CHAR;  }  if (strcmp(buf, "continue") == 0)  {  token1.subType = KW\_CONTINUE;  }  }  else  {  if (containsString(&identifiers, buf))  {  token1.lexeme = findString(&identifiers, buf);  token1.type = TOKEN\_IDENTIFIER;  }  else  {  addString(&identifiers, buf);  token1.lexeme = findString(&identifiers, buf);  token1.type = TOKEN\_IDENTIFIER;  }  }  token1.position.column = nColumn;  nColumn += strlen(buf);  token1.position.row = nRow;  addToken(&tokens, token1);  }  }  else if (isdigit(\*p))  {  p = readNumder(buf, p, bufLen);  bool isFloat = false;  struct Token token1 = {0};  if (p == NULL)  {  printf("\n");  printf("error: incomplete fraction\n\n");  wasError = true;  break;  }  if (\*p != '\0')  {  for (int i = 0; i < strlen(buf); i++)  {  if (buf[i] == '.')  {  isFloat = true;  }  }  if (isFloat == true)  {  if (containsString(&literals, buf))  {  token1.lexeme = findString(&literals, buf);  token1.type = TOKEN\_LITERAL;  token1.subType = LIT\_FLOAT;  }  else  {  addString(&literals, buf);  token1.lexeme = findString(&literals, buf);  token1.type = TOKEN\_LITERAL;  token1.subType = LIT\_FLOAT;  }  }  else  {  if (containsString(&literals, buf))  {  addString(&literals, buf);  token1.lexeme = findString(&literals, buf);  token1.type = TOKEN\_LITERAL;  token1.subType = LIT\_INTEGER;  }  else  {  addString(&literals, buf);  token1.lexeme = findString(&literals, buf);  token1.type = TOKEN\_LITERAL;  token1.subType = LIT\_INTEGER;  }  }  token1.position.column = nColumn;  nColumn += strlen(buf);  token1.position.row = nRow;  addToken(&tokens, token1);  }  }  else if (\*p == '\"')  {  struct Token token1 = {0};  p = readString(buf, p, bufLen);  if (p == NULL)  {  printf("\n");  printf("error: wrong escape sequence\n\n");  wasError = true;  break;  }  if (\*p != '\0')  {  if (containsString(&literals, buf))  {  token1.lexeme = findString(&literals, buf);  token1.type = TOKEN\_LITERAL;  token1.subType = LIT\_STRING;  }  else  {  addString(&identifiers, buf);  token1.lexeme = findString(&identifiers, buf);  token1.type = TOKEN\_LITERAL;  token1.subType = LIT\_STRING;  }  token1.position.column = nColumn;  nColumn += strlen(buf);  token1.position.row = nRow;  addToken(&tokens, token1);  }  }  else if (\*p == ',' || \*p == ';' || \*p == '(' || \*p == ')' || \*p == '=' || \*p == '-'|| \*p == '/')  {  struct Token token1 = {0};  p = reaOpOrDel(buf, p, bufLen);  if (containsString(&DelOrOpLength, buf))  {  token1.lexeme = findString(&DelOrOpLength, buf);  if (strcmp(buf, ",") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_COMMA;  }  if (strcmp(buf, ";") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_SEMICOLON;  }  if (strcmp(buf, "(") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_LEFTPAR;  }  if (strcmp(buf, ")") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_RIGTHPAR;  }  if (strcmp(buf, "=") == 0)  {  token1.type = TOKEN\_OPERATOR;  token1.subType = OP\_ASSIGNMENT;  }  if (strcmp(buf, "-") == 0)  {  token1.type = TOKEN\_OPERATOR;  token1.subType = OP\_SUBSTRACT;  }  if (strcmp(buf, "/") == 0)  {  token1.type = TOKEN\_OPERATOR;  token1.subType = OP\_DIVISION;  }  }  else  {  addString(&DelOrOpLength, buf);  token1.lexeme = findString(&DelOrOpLength, buf);  if (strcmp(buf, ",") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_COMMA;  }  if (strcmp(buf, ";") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_SEMICOLON;  }  if (strcmp(buf, "(") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_LEFTPAR;  }  if (strcmp(buf, ")") == 0)  {  token1.type = TOKEN\_DELIMITER;  token1.subType = DEL\_RIGTHPAR;  }  if (strcmp(buf, "=") == 0)  {  token1.type = TOKEN\_OPERATOR;  token1.subType = OP\_ASSIGNMENT;  }  if (strcmp(buf, "-") == 0)  {  token1.type = TOKEN\_OPERATOR;  token1.subType = OP\_SUBSTRACT;  }  if (strcmp(buf, "/") == 0)  {  token1.type = TOKEN\_OPERATOR;  token1.subType = OP\_DIVISION;  }  }  token1.position.column = nColumn;  nColumn += strlen(buf);  token1.position.row = nRow;  addToken(&tokens, token1);  }  else  {  printf("\n");  printf("error: unknown char\n\n");  wasError = true;  break;  }  if (p == NULL)  {  break;  }  }  if (wasError == false)  {  if (outputFile == false && tokenList == false)  {  for (int i = 0; i < tokens.count; i++)  {  if (aTokens[i].type == TOKEN\_KEYWORD)  Console\_setCursorAttribute(FG\_BLACK);  if (aTokens[i].type == TOKEN\_IDENTIFIER)  Console\_setCursorAttribute(FG\_RED);  if (aTokens[i].type == TOKEN\_DELIMITER)  Console\_setCursorAttribute(FG\_MAGENTA);  if (aTokens[i].type == TOKEN\_OPERATOR)  Console\_setCursorAttribute(FG\_GREEN);  if (aTokens[i].type == TOKEN\_LITERAL)  Console\_setCursorAttribute(FG\_BLUE);  printf("%s", aTokens[i].lexeme);  }  printf("\n");  }  else if (outputFile == false && tokenList == true)  {  printTokens(&tokens);  }  else if (outputFile == true && tokenList == true)  {  FILE \*fw;  fw = freopen(pOutput, "w", stdout);  if (fw == NULL)  {  printf("Error opening file %s\n", pOutput);  exit(EXIT\_FAILURE);  }  printTokens(&tokens);  fclose(fw);  }  }  }  const char \*readWord(char \*dest, const char \*src, int destLen)  {  const char \*p = src;  int counter = 0;  while (isalnum(\*p) || \*p == '\_')  {  if (++counter >= destLen)  {  return (NULL);  }  \*dest = \*p;  dest++;  p++;  }  \*dest = '\0';  return (p);  }  const char \*readNumder(char \*dst, const char \*src, int destLen)  {  int counter = 0;  const char \*p = src;  while (isdigit(\*p))  {  \*dst = \*p;  dst++;  p++;  if (++counter >= destLen)  {  return NULL;  }  }  if (\*p == '.')  {  \*dst = \*p;  dst++;  p++;  if (++counter >= destLen)  {  return NULL;  }  if (!isdigit(\*p))  {  return NULL;  }  else  {  while (isdigit(\*p))  {  if (++counter >= destLen)  {  return NULL;  }  \*dst = \*p;  dst++;  p++;  }  }  }  \*dst = '\0';  return (p);  }  const char \*readString(char \*dst, const char \*src, int destLen)  {  int counter = 0;  const char \*p = src;  p++;  while (\*p != '\"')  {  if (++counter >= destLen)  {  return NULL;  }  if (\*p != '\\')  {  \*dst = \*p;  dst++;  p++;  }  else  {  if (\*(p + 1) == 'n')  {  \*dst = \*p;  dst++;  p++;  }  else  {  return NULL;  }  }  }  p++;  \*dst = '\0';  return p;  }  const char \*reaOpOrDel(char \*dst, const char \*src, int destLen)  {  const char \*p = src;  \*dst = \*p;  dst++;  p++;  \*dst = '\0';  return p;  }  const char \*readWhiteSpace(char \*dst, const char \*src, int destLen)  {  const char \*p = src;  int counter = 0;  if (\*p == '\n' || \*p == '\t')  {  \*dst = \*p;  dst++;  p++;  \*dst = '\0';  }  else  {  while (\*p == ' ')  {  if (++counter >= destLen)  {  return (NULL);  }  \*dst = \*p;  dst++;  p++;  }  \*dst = '\0';  }  \*dst = '\0';  return p;  } |

**Приклади результатів**

Режим виводу списку токенів:

****

Стандартний режим роботи:

****

**Висновок**

Протягом виконання цієї лабораторної роботи я ознайомилася із програмуванням на мові С у спеціалізованому редакторі коду.

Більше того, під час виконання лабораторної роботи я навчилася працювати з текстовими файловими потоками.

Компіляція всього коду відбувалася за допомогою компілятора gcc.

Отже, мета лабораторної роботи досягнена, всі завдання виконані і їх розвʼязання наведені.