rr

**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 №8**

**з дисципліни “ Основи програмування ”**

**тема “СТРУКТУРИ ДАНИХ”**

|  |  |  |
| --- | --- | --- |
| **Виконав**  **студент I курсу**  **групи КП-83**  **Мортіков Владислав Євгенович**  **варіант №8** |  | **Перевірив**  **“\_\_\_\_” “\_\_\_\_\_\_\_\_\_\_\_\_” 20\_\_\_ р.**  **викладач**  **Гадиняк Руслан Анатолійович**  **(прізвище, ім’я, по батькові)** |

**Київ 2018**

**Мета роботи**

Навчитися працювати з рядками за допомогою вказівників. Навчитись програмувати кінцеві автомати для обробки рядків.

**Постановка завдання**

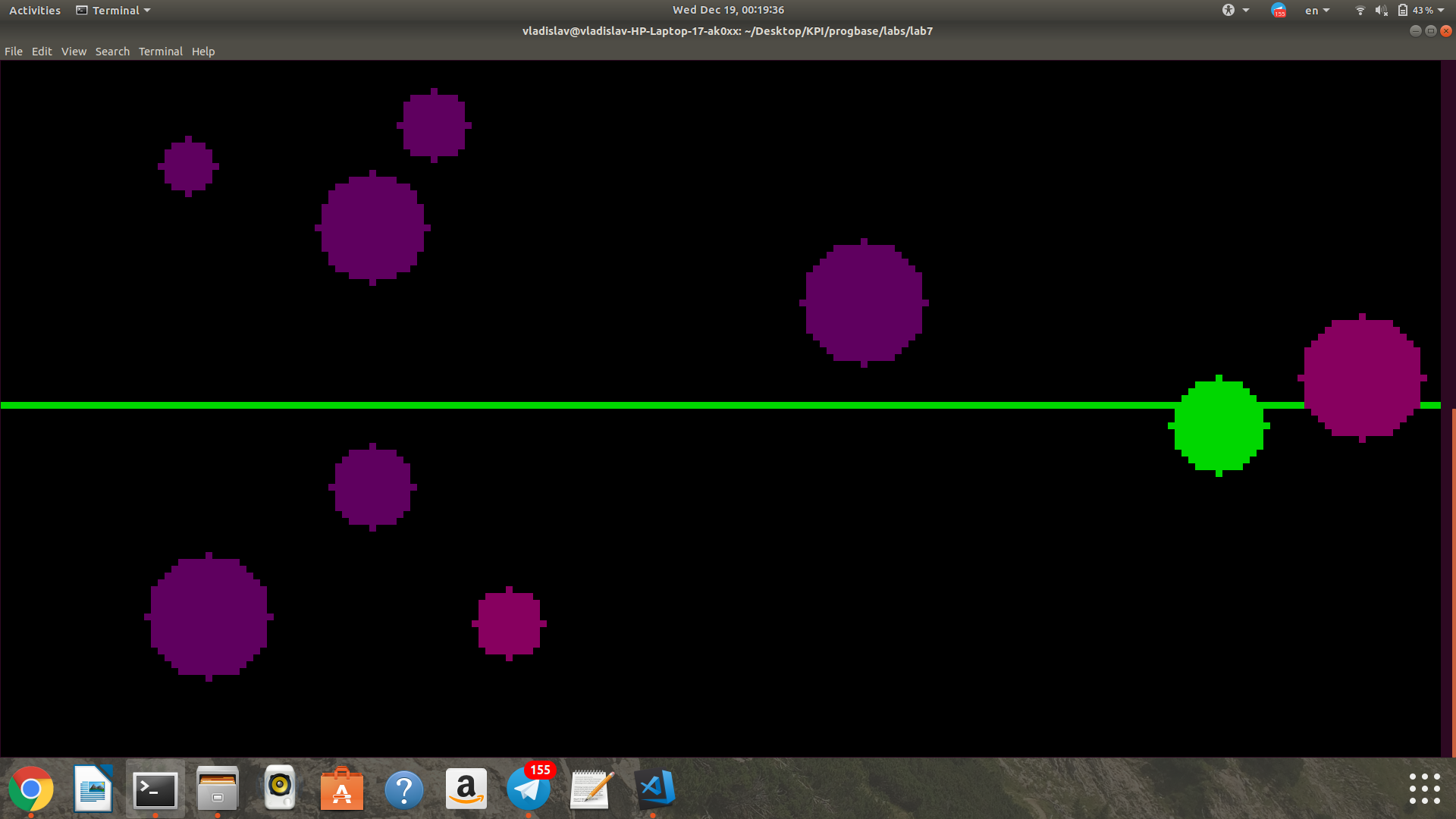
Дано рядок з фрагментом коду мовою С:  
  
 int x = 5 + 5.6 / 2;  
 if (x > 10) {  
 puts("Greater\n");  
 } else if (x < 0) {  
 int a = 42;  
 printf("%d", x - a);  
 }  
 Розбити заданий рядок на список токенів. Кожен токен повинен містити рядок з лексемою та тип токена. Вивести заданий рядок та список токенів у консоль у заданому форматі.  
 Типи токенів:  
Ключове слово (keyword) - одне із зарезервованих слів мови C.  
Оператори (operators) - всі послідовності символів, що формують будь-які оператори мови С (в даному завданні лише односимвольні оператори).  
Розділювачі (delimiters) - спеціальні символи мови:  
 , - кома (comma)  
 ; - крапка з комою (semicolon)  
 ( та ) - круглі дужки (parentheses)  
 { та } - фігурні дужки (curly braces)  
 Ідентифікатор (identifier) - за правилами формування ідентифікаторів мови C. Константний літерал (literal) - значення в коді С, одне із:

Ціле число - непуста послідовність цифр (у даному завданні лише десяткових)  
 Дробове число - непуста послідовність десяткових цифр із однією точкою між цифр (у даному завданні лише повний запис, тобто без неповних і експоненціальних записів)  
 Рядковий літерал - послідовність символів між двох подвійних лапок (послідовність може бути пустою). Також допустимі деякі escape-послідовності.

**Тексти коду програм**

|  |
| --- |
| **main.c** |
| #include <stdio.h>  #include <math.h>  #include <stdlib.h>  #include <progbase.h>  #include <progbase/console.h>  #include <string.h>  #include <ctype.h>  #include <stdbool.h>  enum TokenType  {  TOKEN\_KEYWORD,  TOKEN\_IDENTIFIER,  TOKEN\_LITERAL,  TOKEN\_OPERATOR,  TOKEN\_DELIMITER,  };  enum TokenKeywords  {  KW\_CONST,  KW\_FLOAT,  KW\_BREAK,  };  enum TokenLiterals  {  LIT\_INTEGER,  LIT\_FLOAT,  LIT\_STRING,  };  enum TokenOperators  {  OP\_PLUS,  OP\_MULT,  OP\_ASSIGNMENT,  OP\_DIVISION,  };  enum TokenDelimiters  {  DEL\_LEFTPAR,  DEL\_RIGHTPAR,  DEL\_COMMA,  DEL\_SEMICOLON,  };  struct StringTable  {  char \*items;  size\_t capacity;  size\_t count;  size\_t rowCapacity;  };  struct TokenList  {  struct Token \*items;  size\_t capacity;  size\_t count;  };  struct Token  {  char \*lexeme;  enum TokenType type;  int subType;  };  struct StringTable createStringTable(char \*items, int cap, int rowCap);  struct TokenList createTokenList(struct Token \*a, int n);  void addToken(struct TokenList \*plist, struct Token newToken);  void addString(struct StringTable \*ptable, const char \*str);  char \*getStringAt(const struct StringTable \*ptable, int index);  void printToken(struct TokenList \*plist);  void parseCode(char \*p, struct StringTable \*keywords, struct StringTable \*identifiers, struct StringTable \*literals, struct StringTable \*operators, struct StringTable \*delimiters, struct TokenList \*plist);  char \*readWord(char \*str, char \*dest, const int destSize);  char \*readNumber(char \*str, char \*dest, const int destSize, int \*Dot);  char \*readString(char \*str, char \*dest, const int destSize);  bool IsOperator(char ch, struct Token \*token);  bool IsDelimiter(char ch, struct Token \*token);  int main()  {  Console\_clear();  //start  char string[] = "int x = 5 + 5.6 / 2;\nif (x > 10)\n{\nputs(\"Greater\n\");\n}\n else if (x < 0)\n{\nint a = 42;\nprintf(\"%d\", x - a);\n}";  const int nTokens = 100;  struct Token Tokens[nTokens];  struct TokenList tokens = createTokenList(&Tokens[0], nTokens);  const int nRows = 10;  const int nColoms = 33;  char nKeywords[nRows][nColoms];  char nIdentifiers[nRows][nColoms];  char nLiterals[nRows][nColoms];  char nOperators[nRows][nColoms];  char nDelimiters[nRows][nColoms];  struct StringTable keywords = createStringTable(&nKeywords[0][0], nRows, nColoms);  struct StringTable identifiers = createStringTable(&nIdentifiers[0][0], nRows, nColoms);  struct StringTable literals = createStringTable(&nLiterals[0][0], nRows, nColoms);  struct StringTable operators = createStringTable(&nOperators[0][0], nRows, nColoms);  struct StringTable delimiters = createStringTable(&nDelimiters[0][0], nRows, nColoms);  char \*p = &string[0];  puts("Code >>>");  puts(string);  puts("<<<\n");  parseCode(p, &keywords, &identifiers, &literals, &operators, &delimiters, &tokens);  puts("Tokens >>>");  printToken(&tokens);  puts("<<<\n");  // end  return 0;  }  //create string's list  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;  }  //create token's list  struct TokenList createTokenList(struct Token \*items, int cap)  {  struct TokenList list;  list.items = items;  list.capacity = cap;  list.count = 0;  return list;  }  //add token to list  void addToken(struct TokenList \*plist, struct Token newToken)  {  int presentCount = (\*plist).count;  int index = presentCount;  (\*plist).items[index] = newToken;  int newCount = presentCount + 1;  (\*plist).count = newCount;  }  //add string to list  void addString(struct StringTable \*ptable, const char \*str)  {  int rowIndex = (\*ptable).count;  char \*p = getStringAt(ptable, rowIndex);  strcpy(p, str);  (\*ptable).count += 1;  }  //get strings  char \*getStringAt(const struct StringTable \*ptable, int index)  {  char \*p = (\*ptable).items;  p += index \* (\*ptable).rowCapacity;  return p;  }  //print tokens to list  void printToken(struct TokenList \*plist)  {  for (int i = 0; i < (\*plist).count; i++)  {  if ((\*plist).items[i].type == TOKEN\_KEYWORD)  {  printf("TOKEN\_KEYWORD\t\t");  if ((\*plist).items[i].subType == KW\_CONST)  {  printf("KW\_CONST\t");  }  else if ((\*plist).items[i].subType == KW\_FLOAT)  {  printf("KW\_FLOAT\t");  }  else if ((\*plist).items[i].subType == KW\_BREAK)  {  printf("KW\_BREAK\t");  }  printf("\"%s\"\n", (\*plist).items[i].lexeme);  }  else if ((\*plist).items[i].type == TOKEN\_IDENTIFIER)  {  printf("TOKEN\_IDENTIFIER\t\t\t");  printf("\"%s\"\n", (\*plist).items[i].lexeme);  }  else if ((\*plist).items[i].type == TOKEN\_LITERAL)  {  printf("TOKEN\_LITERAL\t\t");  if ((\*plist).items[i].subType == LIT\_INTEGER)  {  printf("LIT\_INTEGER\t");  printf("\"%s\"\n", (\*plist).items[i].lexeme);  }  else if ((\*plist).items[i].subType == LIT\_FLOAT)  {  printf("LIT\_FLOAT\t");  printf("\"%s\"\n", (\*plist).items[i].lexeme);  }  else if ((\*plist).items[i].subType == LIT\_STRING)  {  printf("LIT\_STRING\t");  printf("%s\n", (\*plist).items[i].lexeme);  }  }  else if ((\*plist).items[i].type == TOKEN\_DELIMITER)  {  printf("TOKEN\_DELIMITER\t\t");  if ((\*plist).items[i].subType == DEL\_COMMA)  {  printf("DEL\_COMMA\t");  }  else if ((\*plist).items[i].subType == DEL\_SEMICOLON)  {  printf("DEL\_SEMICOLON\t");  }  else if ((\*plist).items[i].subType == DEL\_LEFTPAR)  {  printf("DEL\_LEFTPAR\t");  }  else if ((\*plist).items[i].subType == DEL\_RIGHTPAR)  {  printf("DEL\_RIGHTPAR\t");  }  printf("\"%s\"\n", (\*plist).items[i].lexeme);  }  else if ((\*plist).items[i].type == TOKEN\_OPERATOR)  {  printf("TOKEN\_OPERATOR\t\t");  if ((\*plist).items[i].subType == OP\_MULT)  {  printf("OP\_MULT\t\t");  }  else if ((\*plist).items[i].subType == OP\_PLUS)  {  printf("OP\_PLUS\t\t");  }  else if ((\*plist).items[i].subType == OP\_ASSIGNMENT)  {  printf("OP\_ASSIGNMENT\t");  }  else if ((\*plist).items[i].subType == OP\_DIVISION)  {  printf("OP\_DIVISION\t");  }  printf("\"%s\"\n", (\*plist).items[i].lexeme);  }  else  {  printf("List is empty");  }  }  }  // parse code  void parseCode(char \*p, struct StringTable \*keywords, struct StringTable \*identifiers, struct StringTable \*literals, struct StringTable \*operators, struct StringTable \*delimiters, struct TokenList \*plist)  {  const int bufSize = 10;  char buf[bufSize];  int Dot = 0;  struct Token newToken;  while (\*p != '\0')  {  if (isspace(\*p))  {  //ignore  p += 1;  }  else if (isalpha(\*p) || \*p == '\_')  {  p = readWord(p, buf, bufSize);  if (p != NULL)  {  if (strcmp(buf, "const") == 0)  {  addString(keywords, buf);  newToken.lexeme = getStringAt(keywords, (\*keywords).count - 1);  newToken.type = TOKEN\_KEYWORD;  newToken.subType = KW\_CONST;  addToken(plist, newToken);  }  else if (strcmp(buf, "float") == 0)  {  addString(keywords, buf);  newToken.lexeme = getStringAt(keywords, (\*keywords).count - 1);  newToken.type = TOKEN\_KEYWORD;  newToken.subType = KW\_FLOAT;  addToken(plist, newToken);  }  else if (strcmp(buf, "break") == 0)  {  addString(keywords, buf);  newToken.lexeme = getStringAt(keywords, (\*keywords).count - 1);  newToken.type = TOKEN\_KEYWORD;  newToken.subType = KW\_BREAK;  addToken(plist, newToken);  }  else  {  addString(identifiers, buf);  newToken.lexeme = getStringAt(identifiers, (\*identifiers).count - 1);  newToken.type = TOKEN\_IDENTIFIER;  addToken(plist, newToken);  }  }  }  else if (isdigit(\*p))  {  p = readNumber(p, buf, bufSize, &Dot);  if (p != NULL)  {  if (Dot == 1)  {  addString(literals, buf);  newToken.lexeme = getStringAt(literals, (\*literals).count - 1);  newToken.type = TOKEN\_LITERAL;  newToken.subType = LIT\_FLOAT;  addToken(plist, newToken);  }  else  {  addString(literals, buf);  newToken.lexeme = getStringAt(literals, (\*literals).count - 1);  newToken.type = TOKEN\_LITERAL;  newToken.subType = LIT\_INTEGER;  addToken(plist, newToken);  }  }  }  else if (\*p == '"')  {  p = readString(p, buf, bufSize);  if (p != NULL)  {  addString(literals, buf);  newToken.lexeme = getStringAt(literals, (\*literals).count - 1);  newToken.type = TOKEN\_LITERAL;  newToken.subType = LIT\_STRING;  addToken(plist, newToken);  }  }  else if (IsOperator(\*p, &newToken))  {  if (bufSize < 2)  {  p = NULL;  }  else  {  buf[0] = \*p;  buf[1] = '\0';  addString(operators, buf);  newToken.lexeme = getStringAt(operators, (\*operators).count - 1);  newToken.type = TOKEN\_OPERATOR;  addToken(plist, newToken);  p++;  }  }  else if (IsDelimiter(\*p, &newToken))  {  if (bufSize < 2)  {  p = NULL;  }  else  {  buf[0] = \*p;  buf[1] = '\0';  addString(delimiters, buf);  newToken.lexeme = getStringAt(delimiters, (\*delimiters).count - 1);  newToken.type = TOKEN\_DELIMITER;  addToken(plist, newToken);  p++;  }  }  else  {  p += 1;  }  if (p == NULL)  {  printf("Error:\n");  break;  }  }  }  //read word  char \*readWord(char \*str, char \*dest, const int destSize)  {  char \*p = str;  int count = 0;  while (isalnum(\*p) || \*p == '\_')  {  \*dest = \*p;  dest++;  count++;  if (count >= destSize)  {  return NULL;  }  p++;  }  \*dest = '\0';  return p;  }  //indetifier numbers  char \*readNumber(char \*str, char \*dest, const int destSize, int \*Dot)  {  char \*p = str;  int count = 0;  \*Dot = 0;  while (isdigit(\*p))  {  \*dest = \*p;  dest++;  count++;  if (count >= destSize)  {  return NULL;  }  p++;  }  if (\*p == '.')  {  \*Dot = 1;  \*dest = \*p;  dest++;  count++;  if (count >= destSize)  {  return NULL;  }  p++;  if (!isdigit(\*p))  {  return NULL;  }  else  {  while (isdigit(\*p))  {  \*dest = \*p;  dest++;  count++;  if (count >= destSize)  {  return NULL;  }  p++;  }  }  }  \*dest = '\0';  return p;  }  //ident strings  char \*readString(char \*str, char \*dest, const int destSize)  {  char \*p = str;  int count = 0;  \*dest = \*p;  dest++;  p++;  count++;  while (\*p != '"')  {  if (isgraph(\*p) || isspace(\*p))  {  \*dest = \*p;  dest++;  count++;  if (count >= destSize)  {  return NULL;  }  p++;  }  else  {  return NULL;  }  }  \*dest = \*p;  dest++;  p++;  \*dest = '\0';  return p;  }  //operators + - \* / < > =  bool IsOperator(char ch, struct Token \*token)  {  if (ch == '+')  {  token->subType = OP\_PLUS;  }  else if (ch == '\*')  {  token->subType = OP\_MULT;  }  else if (ch == '/')  {  token->subType = OP\_DIVISION;  }  else if (ch == '=')  {  token->subType = OP\_ASSIGNMENT;  }  else  {  return false;  }  return true;  }  //delimeters  bool IsDelimiter(char ch, struct Token \*token)  {  if (ch == ';')  {  token->subType = DEL\_SEMICOLON;  }    else if (ch == ',')  {  token->subType = DEL\_COMMA;  }  else if (ch == '(')  {  token->subType = DEL\_LEFTPAR;  }  else if (ch == ')')  {  token->subType = DEL\_RIGHTPAR;  }  else  {  return false;  }  return true;  } |

**￼Приклади результатів**

****

**Висновки**

Виконавши дану лабораторну роботу ми навчилися працювати з рядками за допомогою вказівників та програмувати кінцеві автомати для обробки рядків.

Компіляція всього коду відбувалася за допомогою компілятора gcc.