

**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 “ Вказівники та обробка рядків ”**

|  |  |  |
| --- | --- | --- |
| **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 з рядками за допомогою вказівників.  
Навчитись програмувати кінцеві автомати для обробки рядків.

**Постановка завдання**

Дано рядок з фрагментом коду мовою С:

char ach\_ = 56;

int k = ach\_ - 10;

printf("%c\t%i something:", ach\_, k / 2);

continue;

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 (див. Додатки).

Типи токенів:

1. Ключове слово (keyword) - одне із зарезервованих слів мови C.
2. Оператори (operators) - всі послідовності символів, що формують будь-які оператори мови С (в даному завданні лише односимвольні оператори).
3. Розділювачі (delimiters) - спеціальні символи мови:
   * , - кома (comma)
   * ; - крапка з комою (semicolon)
   * ( та ) - круглі дужки (parentheses)
   * { та } - фігурні дужки (curly braces)
4. Ідентифікатор (identifier) - за правилами формування ідентифікаторів мови C.
5. 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.

### Методичні рекомендації

Типи даних:

1. Представити у вигляді перечислення (enum):
   1. Типи токенів
   2. Для кожного з типів токенів (окрім типу Ідентифікатор), створити власні перечислення:
      1. Тип ключового слова
      2. Тип оператора
      3. Тип літералу
      4. Тип розділювача
2. Токени описати за допомогою структур (struct), що містять:
   1. Вказівник на початок масиву, що містить рядок із лексемою (рядок матриці символів, таблиця лексем).
   2. Поле типу перечислення, що зберігає тип токена.
   3. Цілочисельне поле для додаткової інформації про токен (тип ключового слова, оператора, літерала або розділювача, див. приклад у Додатках)

Таблиця лексем - матриця символів для зберігання унікальних рядків лексем.

Створити 3 таблиці для лексем:

1. keywords (на (nKeywords)x(nMaxKwLength + 1) - заповнена ключовими словами мови С за варіантом.
2. identifiers (на (nMaxIdentifiers)x(nMaxIdLength + 1) елементів) - для зберігання знайдених унікальних ідентифікаторів із вхідного рядка. Змінна nIdentifiers визначає заповнення таблиці.
3. literals (на (nMaxLiterals)x(nMaxLitLength + 1) елементів) - для зберігання знайдених унікальних літералів із вхідного рядка. Змінна nLiterals визначає заповнення таблиці.

Заповнення та використання таблиць лексем:

1. keywords заповнити перед початком розбору вхідного рядка.
2. Зчитавши ключове слово потрібно знайти посилання на відповідний рядок із таблиці keywords і зберегти його у вказівник токена.
3. Зчитавши ідентифікатор або літерал потрібно додати його лексему у відповідну таблицю лексем, а адресу відповідного рядка матриці символів записати у вказівник токена.

Список токенів:

1. Реалізувати на основі масиву структур фіксованої довжини (масив tokens на nMaxTokens елементів).
2. У такому масиві спочатку повинні міститися структури, що є "заповненими" і відображають елементи списку.
3. Всі інші елементи масиву не рахуються елементами списку.
4. Кількість елементів списку зберігати у окремій цілочисельній змінній nTokens.

Робота зі строками:

1. Для сканування і розбиття вхідного рядка використати алгоритм на основі кінцевого автомата (КА, див. Додатки). Основний КА розбити на простіші КА за допомогою функцій.
2. Кожен символ вхідного рядка при його скануванні зчитувати лише один раз.
3. Для порівняння, копіювання та інших дій з рядками використовувати стандартні функції мови С.

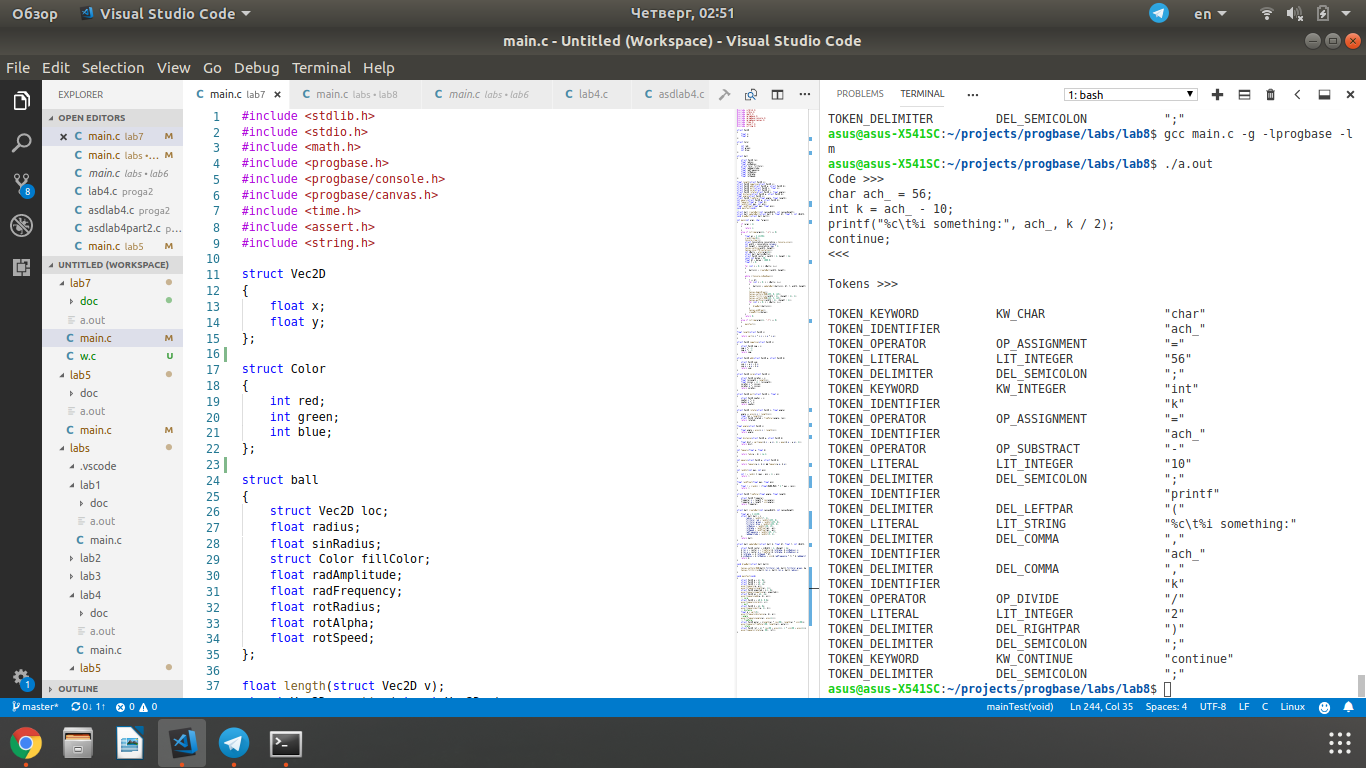
Можливі помилки вхідних даних при зчитуванні:

1. На початку зчитування лексеми зустрівся непустий символ, на який не починається жодна з лексем.
2. Неповний запис дробового числа без дробу.
3. Неіснуюча escape-послідовність у рядковому літералі.

**Тексти коду програм**

|  |
| --- |
| **main.c** |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #include <stdbool.h>  #include <ctype.h>  #include <string.h>  #include <progbase.h>  #include <progbase/console.h>  enum TokenType  {      TOKEN\_KEYWORD,      TOKEN\_IDENTIFIER,      TOKEN\_LITERAL,      TOKEN\_OPERATOR,      TOKEN\_DELIMITER,  };  struct Token  {      char \*lexeme;      enum TokenType type;      int subType;  };  enum TokenKeywords  {      KW\_INTEGER,      KW\_CHAR,      KW\_CONTINUE,  };  enum TokenLiterals  {      LIT\_INTEGER,      LIT\_STRING,  };  enum TokenOperators  {      OP\_DIVIDE,      OP\_SUBSTRACT,      OP\_ASSIGNMENT,  };  enum TokenDelimiters  {      DEL\_LEFTPAR,      DEL\_RIGHTPAR,      DEL\_COMMA,      DEL\_SEMICOLON,  };  struct TokenList  {      struct Token \*tokens;      size\_t nTokens;      size\_t nMaxTokens;  };  struct StringTable  {      char \*items;      size\_t capacity;      size\_t count;      size\_t rowCapacity;  };  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;      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;  }  bool containsString(      const struct StringTable \*ptable,      const char \*str)  {      return findString(ptable, str) != NULL;  }  void output(const struct TokenList \*plist)  {      for (int i = 0; i < (\*plist).nTokens; i++)      {          struct Token t = (\*plist).tokens[i];          if ((\*plist).tokens[i].type == TOKEN\_KEYWORD)          {              printf("TOKEN\_KEYWORD\t\t");              if ((\*plist).tokens[i].subType == KW\_CHAR)              {                  printf("KW\_CHAR\t\t\t");              }              else if ((\*plist).tokens[i].subType == KW\_INTEGER)              {                  printf("KW\_INTEGER\t\t");              }              else if ((\*plist).tokens[i].subType == KW\_CONTINUE)              {                  printf("KW\_CONTINUE\t\t");              }              printf("\"%s\"\n", t.lexeme);          }          else if ((\*plist).tokens[i].type == TOKEN\_LITERAL)          {              printf("TOKEN\_LITERAL\t\t");              if ((\*plist).tokens[i].subType == LIT\_INTEGER)              {                  printf("LIT\_INTEGER\t\t");              }              else if ((\*plist).tokens[i].subType == LIT\_STRING)              {                  printf("LIT\_STRING\t\t");              }              printf("\"%s\"\n", t.lexeme);          }          else if ((\*plist).tokens[i].type == TOKEN\_IDENTIFIER)          {              printf("TOKEN\_IDENTIFIER\t\t\t\t");              printf("\"%s\"\n", t.lexeme);          }          else if ((\*plist).tokens[i].type == TOKEN\_OPERATOR)          {              printf("TOKEN\_OPERATOR\t\t");              if ((\*plist).tokens[i].subType == OP\_ASSIGNMENT)              {                  printf("OP\_ASSIGNMENT\t\t");              }              else if ((\*plist).tokens[i].subType == OP\_SUBSTRACT)              {                  printf("OP\_SUBSTRACT\t\t");              }              else if ((\*plist).tokens[i].subType == OP\_DIVIDE)              {                  printf("OP\_DIVIDE\t\t");              }              printf("\"%s\"\n", t.lexeme);          }          else if ((\*plist).tokens[i].type == TOKEN\_DELIMITER)          {              printf("TOKEN\_DELIMITER\t\t");              if ((\*plist).tokens[i].subType == DEL\_COMMA)              {                  printf("DEL\_COMMA\t\t");              }              else if ((\*plist).tokens[i].subType == DEL\_SEMICOLON)              {                  printf("DEL\_SEMICOLON\t\t");              }              else if ((\*plist).tokens[i].subType == DEL\_LEFTPAR)              {                  printf("DEL\_LEFTPAR\t\t");              }              else if ((\*plist).tokens[i].subType == DEL\_RIGHTPAR)              {                  printf("DEL\_RIGHTPAR\t\t");              }              printf("\"%s\"\n", t.lexeme);          }      }  }  void addToken(struct TokenList \*ptokens, struct Token token)  {      int index = (\*ptokens).nTokens;      (\*ptokens).tokens[index] = token;      (\*ptokens).nTokens += 1;  }  const char \*readNumder(char \*dest, const char \*src, int destLen)  {      const char \*p = src;      int counter = 0;      while (isdigit(\*p))      {          \*dest = \*p;          dest++;          if (++counter >= destLen)          {              return NULL;          }          p++;      }      if (\*p == '.')      {          \*dest = \*p;          dest++;          if (++counter >= destLen)          {              return NULL;          }          p++;          if (!isdigit(\*p))          {              return NULL;          }          while (isdigit(\*p))          {              \*dest = \*p;              dest++;              if (++counter >= destLen)              {                  return NULL;              }              p++;          }      }      \*dest = '\0';      return p;  }  const char \*readWord(char \*dest, const char \*src, int destLen)  {      const char \*p = src;      int counter = 0;      while (isalnum(\*p) || \*p == '\_')      {          \*dest = \*p;          dest++;          if (++counter >= destLen)          {              return NULL;          }          p++;      }      \*dest = '\0';      return p;  }  const char \*readString(char \*dest, const char \*src, int destLen)  {      const char \*p = src;      int counter = 0;      if (\*p == '"')      {          p++;          while (\*p != '"')          {              if (\*p == '\\')              {                  if (\*p != 'n' || \*p != 'r' || \*p != 't')                  {                      \*dest = \*p;                      dest++;                      if (++counter >= destLen)                      {                          return NULL;                      }                      p++;                  }              }              \*dest = \*p;              dest++;              p++;          }          \*dest = '\0';          p++;      }      return p;  }  const char \*readDel(char \*dest, const char \*src, int destLen)  {      const char \*p = src;      int counter = 0;      if (\*p == ';' || \*p == ',' || \*p == '(' || \*p == ')')      {          \*dest = \*p;          dest++;          if (++counter >= destLen)          {              return NULL;          }          p++;      }      \*dest = '\0';      return p;  }  const char \*readOp(char \*dest, const char \*src, int destLen)  {      const char \*p = src;      int counter = 0;      while (\*p == '=' || \*p == '-' || \*p == '/')      {          \*dest = \*p;          dest++;          if (++counter >= destLen)          {              return NULL;          }          p++;      }      \*dest = '\0';      return p;  }  void parseCode(const char \*p)  {      const int nKeywords = 3;      const int nMaxKwLength = 10;      char keywords[nKeywords][nMaxKwLength + 1];      struct StringTable kws = createStringTable(&keywords[0][0], nKeywords, nMaxKwLength);      addString(&kws, "int");      addString(&kws, "char");      addString(&kws, "continue");      //      const int nOp = 3;      const int nMaxOpLength = 10;      char operators[nOp][nMaxOpLength + 1];      struct StringTable ops = createStringTable(&operators[0][0], nOp, nMaxOpLength);      addString(&ops, "-");      addString(&ops, "=");      addString(&ops, "/");      const int nDel = 3;      const int nMaxDelLength = 10;      char delimiters[nDel][nMaxDelLength + 1];      struct StringTable dels = createStringTable(&delimiters[0][0], nDel, nMaxDelLength);      addString(&dels, ",");      addString(&dels, ";");      addString(&dels, "(");      addString(&dels, ")");      //      const int nMaxIdentifiers = 20;      const int nMaxIdLength = 25;      char identifiers[nMaxIdentifiers][nMaxIdLength + 1];      struct StringTable ids = createStringTable(&identifiers[0][0], nMaxIdentifiers, nMaxIdLength);      //      const int nMaxLiterals = 20;      const int nMaxLitLength = 25;      char literals[nMaxLiterals][nMaxLitLength + 1];      struct StringTable lits = createStringTable(&literals[0][0], nMaxLiterals, nMaxLitLength);      // list of structures      const int nMaxTokens = 30;      struct Token tokens[nMaxTokens];      struct TokenList list;      list.tokens = &tokens[0];      list.nMaxTokens = nMaxTokens;      list.nTokens = 0;      //      struct Token newToken;      const int bufLen = 100;      char buf[bufLen];      while (\*p != '\0')      {          if (isalpha(\*p) || \*p == '\_')          {              p = readWord(buf, p, bufLen);              if (p != NULL)              {                  if (containsString(&kws, buf))                  {                      newToken.lexeme = findString(&kws, buf);                      newToken.type = TOKEN\_KEYWORD;                      if (strcmp(buf, "int") == 0)                      {                          newToken.subType = KW\_INTEGER;                      }                      else if (strcmp(buf, "continue") == 0)                      {                          newToken.subType = KW\_CONTINUE;                      }                      else if (strcmp(buf, "char") == 0)                      {                          newToken.subType = KW\_CHAR;                      }                  }                  else                  {                      addString(&ids, buf);                      newToken.lexeme = findString(&ids, buf);                      newToken.type = TOKEN\_IDENTIFIER;                  }                  addToken(&list, newToken);              }          }          else if (isdigit(\*p))          {              p = readNumder(buf, p, bufLen);              if (p != NULL)              {                  addString(&lits, buf);                  newToken.lexeme = findString(&lits, buf);                  newToken.type = TOKEN\_LITERAL;                  newToken.subType = LIT\_INTEGER;                  addToken(&list, newToken);              }          }          else if (\*p == '"')          {              p = readString(buf, p, bufLen);              if (p != NULL)              {                  addString(&lits, buf);                  newToken.lexeme = findString(&lits, buf);                  newToken.type = TOKEN\_LITERAL;                  newToken.subType = LIT\_STRING;                  addToken(&list, newToken);              }          }          else if (\*p == ';' || \*p == ',' || \*p == '(' || \*p == ')')          {              p = readDel(buf, p, bufLen);              if (p != NULL)              {                  if (containsString(&dels, buf))                  {                      newToken.lexeme = findString(&dels, buf);                      newToken.type = TOKEN\_DELIMITER;                      if (strcmp(buf, ",") == 0)                      {                          newToken.subType = DEL\_COMMA;                      }                      else if (strcmp(buf, ";") == 0)                      {                          newToken.subType = DEL\_SEMICOLON;                      }                      else if (strcmp(buf, "(") == 0)                      {                          newToken.subType = DEL\_LEFTPAR;                      }                      else if (strcmp(buf, ")") == 0)                      {                          newToken.subType = DEL\_RIGHTPAR;                      }                  }                  addToken(&list, newToken);              }          }          else if (\*p == '=' || \*p == '-' || \*p == '/')          {              p = readOp(buf, p, bufLen);              if (p != NULL)              {                  if (containsString(&ops, buf))                  {                      newToken.lexeme = findString(&ops, buf);                      newToken.type = TOKEN\_OPERATOR;                      if (strcmp(buf, "=") == 0)                      {                          newToken.subType = OP\_ASSIGNMENT;                      }                      else if (strcmp(buf, "-") == 0)                      {                          newToken.subType = OP\_SUBSTRACT;                      }                      else if (strcmp(buf, "/") == 0)                      {                          newToken.subType = OP\_DIVIDE;                      }                  }                  addToken(&list, newToken);              }          }          else          {              p++;          }          if (p == NULL)          {              printf("\nSome error occured!\n");              break;          }      }      output(&list);  }  int main()  {      char str[100] = "char ach\_ = 56;\nint k = ach\_ - 10;\nprintf(\"%c\\t%i something:\", ach\_, k / 2);\ncontinue;";      //      char \*p = str;      puts("Code >>>");      puts(p);      puts("<<<\n\nTokens >>>\n");      parseCode(p);      return 0;  } |

**Приклади результатів**

****

**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 рядками за допомогою вказівників. Також мною було засвоєно програмування кінцевих автоматів для обробки рядків.

Компіляція всього коду відбувалася за допомогою компілятора gcc.

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