# ­Звіт до лабораторної 3

Студента групи ТТП-32

Остренка Олександра

## 1. Умова лабораторної:

Реалізувати лексичний аналізатор мови програмування C++. Для зберігання класів лексем організувати таблиці. Вивести вміст таблиць після обробки тексту програми.

Розрізняти принаймні такі класи лексем:

* числа (десяткові, з плаваючою крапкою, шістнадцяткові),
* рядкові та символьні константи,
* директиви препроцесора,
* коментарі,
* зарезервовані слова,
* оператори,
* розділові знаки,
* ідентифікатори.

Позначати ситуації з помилками (наприклад, нерозпізнані символи).

Можливі варіанти виводу результату роботи програми:

* “розфарбування” тексту програми (наприклад, кольором),
* вивід пар < лексема , тип\_лексеми >,
* вивести лексеми, що зустрілися в програмі, розбитими на класи.

## 2. Опис виконання:

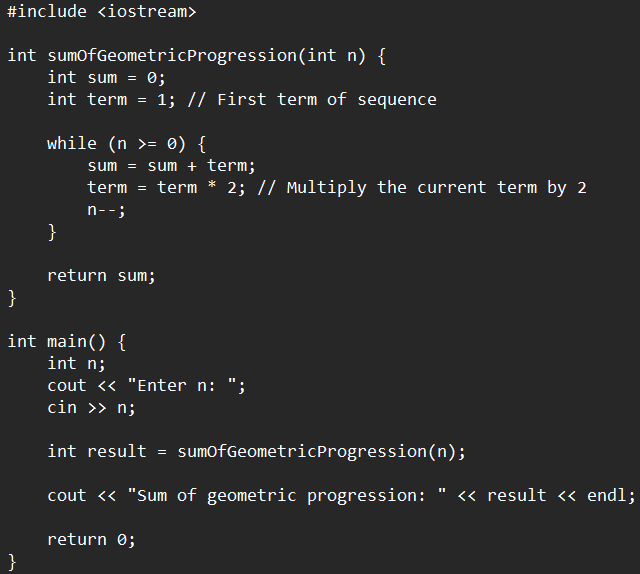
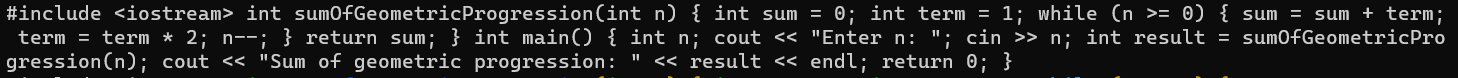
Реалізацію лабораторної можна розбити на етапи:

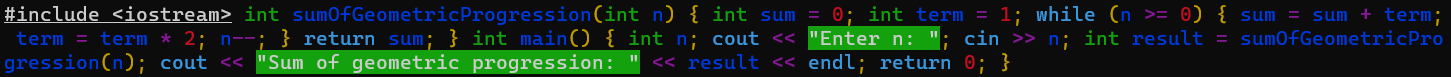
1. Спочатку треба вирішити, яким чином подавати текст в лексер. Автор пропонує зчитувати .txt файл, якому буде відповідний текст. Для цього автор використовує функцію “string ReadFile(string path)”, яка зчитує файл по рядкам і зберігає текст у “string sourceCode;”.
2. Далі треба підготувати текст до обробки. В даному випадку підготовка - це прибирання усього зайвого: коментарів (// ... або /\*...\*/), переносів строки (/n), або ж непотрібних пробілів. Для виконання даного етапу, автор використовує функцію “string prepareTextForLA(const string& inputText)”.  
   Функція відпрацьовує наступним чином: за допомогою регулярних виразів, шукає та заміняє знайдені коментарі на “”, послідовності пробілів та переносів строки - на один пробіл:
   1. Для пошуку коментарів, автор використовує наступний регулярний вираз: “//.\*|/\\\*.\*?\\\*/”,   
      ”//.\*” означає, що ми виділяємо усі слова, що йдуть після “//” і до кінця строки,  
      ”/\\\*.\*?\\\*/” означає, що ми виділяємо усі слова, що починаються на “/\*” (\\ потрібен перед \*, т.я \* також є службовим символом в регулярках ), та закінчуються на “\*/” (знак питання означає що пошук буде не жадібним, тобто зі строки “/\*...\*/.../\*...\*/” він виділить на два слова, а не одне).
   2. Для пошуку та видалення всіх зайвих пробілів (та переносів строки) використовуємо регулярний вираз: “\\s+”, де “s” позначає пробіл, а “+” - "один чи більше"
   3. Т.я. при видаленні зайвих пробілів ми заміняємо усі на один, то на початку та в кінці може залишитись по пробілу, тож треба їх видалити: “^\\s+|\\s+$”.
3. Після того, як текст підготовлено, залишається пройтись текстом лексером, та виділити відповідні лексеми. Автором було вирішено розфарбувати усі лексеми відповідно до кольорів їх класу. Автор виділив наступні класи:
   1. Директиви предпроцессора підкреслюються білою рискою. Щоб виділити усі директиви #include, автор використовує наступний регулярний вираз: “include <.\*?>”.
   2. Позначення типів (int, double, …) автор виділяє зеленим кольором, та використовує регулярний вираз “\\b(int|float|double|char)\\b” задля їх пошуку. Зарезервована конструкція “\\b(...)\b” означає, що ми шукаємо повноцінне слово, а не слово, що має те слово як підрядок.
   3. Числа автор виділяє червоним кольором, та використовує регулярний вираз “\\b\\d+(\\.\\d+)?\\b” задля їх пошуку. “\\d+” - кілька цифр 0-9 підряд, “(\\.\\d+)?” - допускається, що після тих цифр може бути крапка, та ще цифри.
   4. Для рядкових та символьних констант, автор зарисовує фон зеленим, а регулярний вираз виглядає, як “(\".\*?\")|('.\*?')”. “\” використовується щоб екранувати кавичку, бо вона є службовим символом в регулярках.
   5. Щоб позначити ключові слова, автор використовує голубий колір та наступну регулярку - “\\b(if|else|while|return|cout|cin|endl)\\b”.
   6. Для позначення операторів, автор використовує фіолетовий колір та наступний регулярний вираз - “[+\\-\*/=<>]”
   7. Для позначення розділових знаків, автор використовує жовтий колір та наступний регулярний вираз - “[;{}()\\[\\]]”.
   8. Для позначення змінних (послідовностей латинських символів), автор використовує наступний регулярний вираз - “\\b[a-zA-Z]+\\b”

Функція, що фактично є лексером - “void highlightLexemes(const string& code)”, а працює вона наступним чином:

* Автор використовує “map<string, string> regexToColor”, щоб зіставити кожному регулярному виразу відповідний колір.
* Далі, з усіх регулярних виразів у “regexToColor” автор утворює один, комбінований, регулярний вираз “combinedRegex”, а з нього - екземпляр класу для роботи з регулярними виразами “regex lexemeRegex(combinedRegex);” та екземпляр класу для зберігання результатів зіставлення регулярного виразу з текстом “smatch matches;”
* Далі, автор проходиться в циклі по рядку і шукає зіставлення тексту рядка, з якимось з регулярних виразів у lexemеRegex:   
  “ while (regex\_search(code.begin() + position, code.end(), matches, lexemeRegex))”  
  ”code.begin() + position” - точка, з якої треба шукати зіставлення;  
  ”code.end()” - кінцева точка;  
  ”matches” - змінна, в яку зберігається кусок текста з code, який пройшов через сукупність регулярних виразів “lexemeRegex”, (треба відмітити, що текст зіставляється з кожною регуляркою з “lexemeRegex” по-черзі, до поки зіставлення не буде успішним, тому треба обережно слідкувати за їх порядком в сукупності, бо одна може ненароком перекрити інші, та “вкрасти усю славу”);
* Далі, треба вивести на екран усі символи між словом, що пройшло перевірку, та початком аналізу, сукупність тих символів не пройшла через жоден регулярний вираз, тож можемо вважати їх за лексичну помилку, тому ми не виділяємо їх жодним кольором і вони залишаються білими, але вивести їх необхідно.
* Після того останній шаг - розфарбувати відповідну лексему. Щоб зрозуміти, яким саме кольором розфарбувати, скористаємося раніше об`явленим масивом “map<string, string> regexToColor” та, в циклі, знайдемо потрібний колір:  
    
    
  ”for (const auto& entry : regexToColor) {  
   if (regex\_match(matchedText, regex(entry.first))) {”  
  ”regex\_match” потрібен для того, щоб зіставити відповідний регулярний вираз з текстом, що пройшов регулярку. Пройшовши перевірку, відправляємось в функцію, що виведе розфарбований в потрібний колір текст. Функція сама по собі нецікава, тож розглядатись не буде.

## 3. Тестування.

Текст для тестування:   
  
Проміжний результат - текст, після підготовки до лексера.  


І остаточний результат:  


## 4. Посилання на GitHub.

[SashAmlet/Lexical-Analyzer: Lexical analyzer for C# language based on regular expressions (github.com)](https://github.com/SashAmlet/Lexical-Analyzer)