Модуль ввода вывода и лексический анализатор

На данном этапе нам необходимо входной файл разбить на лексемы, в соответствии с синтаксисом языка, а именно его ключевыми словами и специальными символами, определив для каждой лексемы её тип и сохраняя их в определённый буфер и определить лексические ошибки.

Концепция данного модуля заключается на обработке отдельных групп символов. Набор всех допустимых символов, был разбит на 3 подмножества.

1. Множество букв в нижнем и вернем регистре и нижнее подчёркивание
2. Множество цифр
3. Множество символов языка включая составные и ключевые слова

А так же выделены группы специальных слов

Для работы предполагается система считывающего модуля и буфера символов.

Что бы понять, что мы считали мы используем несколько групп ввода, а именно:

1. Ввод числа, целого и вещественного типа
2. Ввод символьной строки
3. Ввод символьной константы
4. Ввод строковой константы
5. Ввод специальных символов

Так же мы должны определить, какие лексические ошибки мы должны определять в компилируемой программе.

Я остановился на следующих ошибках:

1. Запрещенный символ, символ которого нет в языке.
2. Неправильный формат идентификатора, начинается с цифры.
3. Неправильный формат символьной константы, пустая.
4. Превышен лимит константы, размер константы превышает разрешенную длину, для символа длинна 1, для чисел 7.
5. Нарушен формат для вещественного числа, наличие в нём более одной точки например 2..0
6. Нарушена длина идентификатора, длинна идентификатора более 10 символов.

Реализация программы

Первая сложность как реализовать токен, это проблема решилась созданием для него отдельного класса

**class** **CToken**

{

**public:**

TokenType tt;

OperSymbol oper;

Variant\* const\_val = NULL;

std::string ident = "";

CToken(TokenType \_tt, OperSymbol \_op, std::string val) // операция

{

tt = \_tt;

oper = \_op;

const\_val = nullptr;

ident = val;

}

CToken(TokenType \_tt, Variant\* \_constVal, std::string val)

{

tt = \_tt;

oper = none;

const\_val = \_constVal;

ident = val;

}

CToken(TokenType \_tt, std::string val) //идентификатор

{

tt = \_tt;

oper = none;

const\_val = nullptr;

ident = val;

}

~CToken() {

**delete** const\_val;

}

std::string GetTypeToken()

{

**switch** (tt)

{

**case** ttOperation: **return** "Operation";

**case** ttConst: **return** "Const";

**case** ttIdent: **return** "Identifier";

}

}

};

Так же было необходимо понять как хранить константы изначально я хотел использовать кортежи из variant но оно почему то не работало поэтому я принял решение прибегнуть к ООП

**class** **Variant**

{

**public:**

**virtual** std::string GetClassVariant() = **0**;

};

**class** **IntVariant** : **public** Variant

{

**private:**

**int** value;

**public:**

IntVariant(**int** val) { value = val; }

**int** GetValue() { **return** value; }

std::string GetClassVariant() override { **return** "int"; }

};

**class** **FloatVariant** : **public** Variant

{

**private:**

**float** value;

**public:**

FloatVariant(**float** val) { value = val; }

**float** GetValue() { **return** value; }

std::string GetClassVariant() override { **return** "float"; }

};

**class** **BoolVariant** : **public** Variant

{

**private:**

**bool** value;

**public:**

BoolVariant(**bool** val) { value = val; }

**bool** GetValue() { **return** value; }

std::string GetClassVariant() override { **return** " bool"; }

};

**class** **StringVariant** : **public** Variant

{

**private:**

std::string value;

**public:**

StringVariant(std::string val) { value = val; }

std::string GetValue() { **return** value; }

std::string GetClassVariant() override { **return** "string"; }

};

Методы получения констант

Variant\* **GetConstInt**(std::string buf)

{

**int** i = **0**;

**int** const\_value = **0**;

**while** (buf[i] != '.' && i < buf.size())

{

const\_value \*= **10**;

const\_value += (buf[i] - '0');

i++;

}

IntVariant\* a = **new** IntVariant(const\_value);

**return** a;

}

Variant\* **GetConstFloat**(std::string buf)

{

**int** i = **0**;

**float** const\_value = **0**;

**while** (buf[i] != '.' && i < buf.size())

{

const\_value \*= **10**;

const\_value += (buf[i] - '0');

i++;

}

i++;

**int** dr\_val = **0**;

**while** (i < buf.size())

{

dr\_val += (buf[i] - '0');

dr\_val /= **10**;

i++;

}

FloatVariant\* a = **new** FloatVariant(const\_value);

**return** a;

}

Variant\* **GetConstBool**(std::string buf)

{

**bool** const\_value;

**if** (buf == "true")

const\_value = true;

**if** (buf == "false")

const\_value = false;

BoolVariant\* a = **new** BoolVariant(const\_value);

**return** a;

}

Variant\* **GetConstString**(std::string buf)

{

std::string const\_value = "";

**for** (**int** i = **0**; i < buf.size(); i++)

{

**if** (buf[i] != '\'')

{

const\_value += buf[i];

}

}

StringVariant\* a = **new** StringVariant(const\_value);

**return** a;

}

Для инициализации ошибок был создан словарь ошибок с кодами и сообщениями.

Так же возникает одна из самых сложных проблем нейтрализация лексических ошибок

Решил я её довольно просто оператором goto

Так же возникла проблема вывода ошибки так как иногда мы ловим её на этапе чтения как одного символа так и на этапе готового токена

Поэтому я создал отдельную функцию для вывода ошибок и перегрузил её

**void** **GetError**(**int** i, **int** j, std::string buf, **int** code)

{

error A;

A.code\_error = code;

std::pair<**int**, **int**> B;

B.first = i;

B.second = j;

A.XoY = B;

//eror.push\_back(A);

std::cout << "'" << buf << "' " << "Code error : " << A.code\_error << " line: " << A.XoY.second << " position: " << A.XoY.first << std::endl;

}

**void** **GetError**(**int** i, **int** j, **char** buf, **int** code)

{

error A;

A.code\_error = code;

std::pair<**int**, **int**> B;

B.first = i;

B.second = j;

A.XoY = B;

//eror.push\_back(A);

std::cout << "'" << buf << "' " << "Code error : " << A.code\_error << " line: " << A.XoY.second << " position: " << A.XoY.first << std::endl;

}

Чтение лексем я делал через оператор get

**bool** **READ**(std::ifstream& fl, **char**& leks)

{

fl.get(leks);

**if** (!fl)

**return** false;

**return** true;

}

А вот и сам модуль IO и лексер

CToken\* **IO**(std::string buf, std::string& last\_token, **char**& leks, std::ifstream& fl, **int**& i, **int**& j)

{

TokenType token = ttIdent;

**Again:**

**if** (leks == '\n')

{

j++;

}

**while** (leks == '\t' || leks == ' ' || leks == '\n' || leks == '\0') // сббрасываем табы энтеры пустые символы

{

**if** (leks == '\t')

i += **4**;

**else**

i++;

**if** (leks == '\n')

j++;

**if** (!READ(fl, leks))

**return** NULL;

}

**while** (num(leks) || word(leks) && buf!="" || buf != "" && leks == '.') // читаем число

{

token = ttConst;

buf += leks;

**if** (!READ(fl, leks))

{

leks = '\0';

**break**;

}

**if** (leks == '\t')

i += **4**;

**else**

i++;

**if** (leks == '\n')

j++;

}

**if** (buf!="" && !NUM(buf))

{

**bool** flag = false;

**if** (num(buf[**0**]))

{

**for** (**int** x = **0**; x < buf.size(); x++)

{

**if** (word(buf[x]))

{

GetError(i, j, buf, **2**);

buf = "";

**goto** Again;

}

**if** (buf[x] == '.')

{

**if** (flag == true) {

GetError(i, j, buf, **3**);

buf = "";

**goto** Again;

}

flag = !flag;

}

}

}

}

**if** (buf != "" && NUM(buf)) // сбрасываем токен числа

{

**if** (buf.find('\.') != -**1**)

{

**return** **new** CToken(ttConst, GetConstInt(buf), buf);

}

**return** **new** CToken(ttConst, GetConstFloat(buf), buf);

}

**if** (leks == '\"') { // строковая константа

token = ttConst;

**do**

{

i++;

buf += leks;

**if** (!READ(fl, leks)) {

leks = '\0';

**break**;

}

} **while** (leks != '\"');

**if** (leks == '\0')

{

GetError(i, j, buf, **1**);

**return** NULL;

}

buf += leks;

leks = '\0';

**return** **new** CToken(token, GetConstString(buf), buf);

}

**if** (leks == '\'') { // символьная константа

token == ttConst;

**do** {

i++;

buf += leks;

**if** (!READ(fl, leks)) {

leks = '\0';

**break**;

}

} **while** (leks != '\'');

buf += leks;

leks = '\0';

**if** (buf.size() > **2** || buf == "''")

{

GetError(i, j, buf, **1**);

**if** (!READ(fl, leks))

{

**return** NULL;

}

buf = "";

**goto** Again;

}

**return** **new** CToken(token, GetConstString(buf), buf);

}

**while** (word(leks) || num(leks) || leks == '\_') // читаем слово

{

token = ttIdent;

buf += leks;

**if** (!READ(fl, leks)) {

leks = '\0';

**break**;

}

**if** (leks == '\t')

i += **4**;

**else**

i++;

**if** (leks == '\n')

j++;

}

**if** (buf == "bool" || buf == "false")

{

token = ttConst;

**return** **new** CToken(ttConst, GetConstBool(buf), buf);

}

**if** (SpecToken.find(buf) != SpecToken.end())

{

token = ttOperation;

**return** **new** CToken(token, buf);

}

**if** (buf != "") {

**return** **new** CToken(ttIdent, buf);

}

**while** (SpecToken.find(ToString(leks)) != SpecToken.end() && leks != '\t' && leks != '\0' && leks != '\n' && leks != '\'')

{

token = ttOperation;

buf += leks;

**if** (!READ(fl, leks)) {

leks = '\0';

**break**;

}

**if** (leks == '\t')

i += **4**;

**else**

i++;

**if** (leks == '\n')

j++;

}

**if** (buf != "")

{

**return** **new** CToken(token, buf);

}

GetError(i, j, leks, **0**);

**if** (!READ(fl, leks))

{

**return** NULL;

}

**goto** Again;

}