|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Министерство науки и высшего образования РФ | | | | | | | | |
|  | | |  | | |  | | |
| Пермский государственный национальный  исследовательский университет | | | | | | | | |
|  | | |  | | |  | | |
|  | | ОТЧЁТ  по лабораторной работе “Разработка компилятора языка Pascal»  по дисциплине «Формальные грамматики и методы трансляции” | | | | |  | |
|  | | |  | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-1,2  Валеев Р. Р. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_\_ 2021 | | |  | Проверил  асис.кафедры МОВС  Пономарев Ф.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_ 2021 | | |  |
|  |  | | |  |  | | |  |
| Пермь 2021 | | | | | | | | |

# Анализ

Для решения глобальной задачи – написание компилятора для подмножества языка Pascal, требуется решить несколько подзадач, а именно, спроектировать и разработать модуль ввода-вывода, лексический, синтаксический и семантический анализаторы, а также генератор кода. Первым этапом станет создание модуля ввода-вывода и лексического анализатора. Модуль ввода-вывода осуществляет считывание литер из входного файла, а также выдаёт сообщения об ошибках. Лексический анализатор, в свою очередь, переводит последовательность литер в символы и возвращает лексемы.

Перед проектированием было важно детальнее изучить структуру компилятора. Начальные сведения были получены из первых лекций по этому предмету, а также после знакомства с начальными главами книги Залоговой «Разработка Паскаль-компилятора». Ответы на вопросы, возникающие в ходе этапов планирования и разработки, удавалось находить на следующих источниках: <https://docs.microsoft.com/ru-ru/>, [stackoverflow.com](https://stackoverflow.com/), <https://habr.com> и [www.cyberforum.ru](http://www.cyberforum.ru). В основном это был поиск идей для решения той или иной проблемы или прочтение документации о неизвестных мне возможностях языка C#.

# Проектирование

Перед началом разработки было проведено начальное проектирование. На рисунке ниже представлена спроектированная диаграмма классов

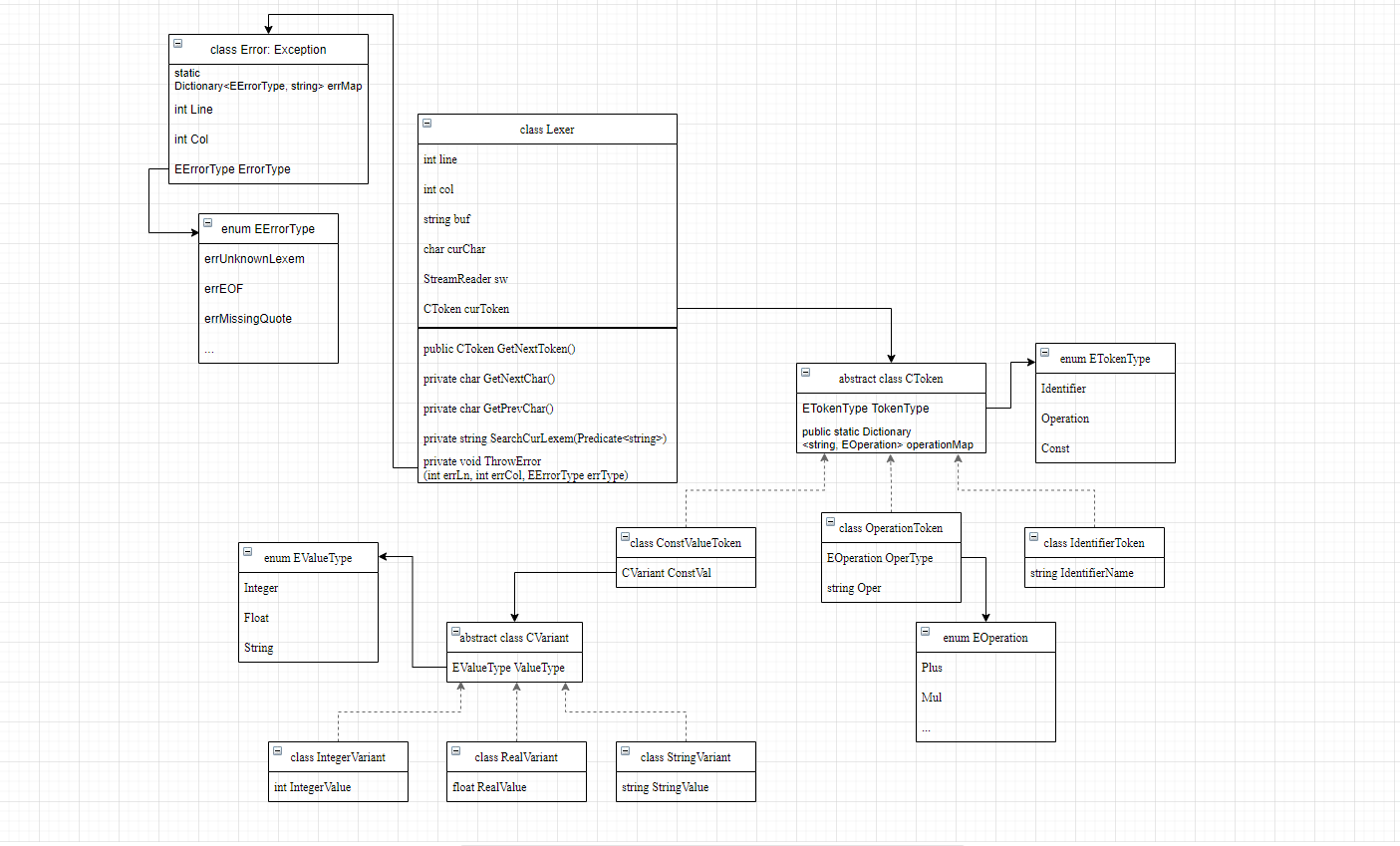


Рисунок 1 – Диаграмма классов

Типы токенов, констант, ошибок, а также все операции и ключевые слова хранятся в виде перечислений. Для сопоставления последовательности литер соответствующему токену используется словарь операций operationMap. Каждому типу токена соответствует собственный класс, который унаследован от абстрактного класса CToken. Информация о токене обязательно включает в себя тип токена. В зависимости от типа токена также может храниться тип операции, значение константы или название идентификатора. Для хранения значения константы был спроектирован класс CVariant, являющийся абстрактным. В нём хранится информация о типе константы. Классы IntegerVariant, RealVariant, StringVariant унаследованы от CVariant и хранят в себе значение требуемого типа.

Лексический анализатор вынесен в отдельный класс Lexer. В данном классе будет хранится информация о последнем полученном токене, текущая литера и её расположение (номер строки и место позиции в строке). Основным методом лексического анализатора является GetNextToken(). Он формирует очередную лексему или возвращает ошибку, возникшую при формировании очередной лексемы. Для выдачи ошибки используется метод ThrowError(), который в качестве входных параметров принимает информацию об ошибке, и возвращает сформированную ошибку.

Модуль ввода-вывода является частью лексического анализатора. Для считывания исходного текста программы используется метод GetNextChar(), который возвращает следующую литеру. Данный метод считывает текст программы построчно, помещая считанную строку в буфер. Как только достигается конец строки, то из исходного файла считывается новая строка.

Информация об ошибках выводится по ходу их обработки. Для ошибок создан отдельный класс Error. Он наследует класс Exception – базовый класс, от которого наследуются исключения. В классе Error хранится информация о типе ошибки и месте её возникновения. Описание ошибки можно получить по её коду. Для этого используется словарь errorMap, где каждому коду ошибки соответствует её уникальное описание.

# Разработка

После окончания этапа проектирования начался этап разработки. Сначала был реализован класс CToken и все его дочерние классы, связанные с хранением различных типов лексем. После чего началась разработка лексического анализатора. Класс Lexer ответственен за перевод последовательности символов в лексемы. В нём реализованы следующие методы: GetNextChar(), GetPrevChar(), SearchCurLexem(), GetNextToken() и ThrowError().

Метод GetNextChar() относится к модулю ввода-вывода. Он делает следующую литеру из буфера текущей. Если при считывании очередной литеры достигается конец строки буфера, то из файла считывается новая строка и помещается в буфер. Также полученная строка сразу выводится в выходной файл. Реализация данного метода представлена ниже:

private char GetNextChar()

{

if (col == buf.Length)

{

string str = sr.ReadLine();

if (str != null)

{

Program.sw.WriteLine($"{line + 1,4}. {str}");

buf = str + '\n';

line++;

col = 0;

}

else

buf += "\0";

}

return buf[col++];

}

Метод ThrowError() выдаёт ошибку и осуществляет переход к следующей литере. В качестве входных параметров передаются номер строки и номер позиции, в которой произошла ошибка, а также тип ошибки.

Метод GetPrevChar() делает предыдущую литеру из буфера текущей. Данный метод используется в тех случаях, когда для распознавания лексемы требуется знать две подряд идущих литеры. Это те случаи, когда первая литера может встречаться в разных символах и вместе со второй литерой образовывать другой символ. Например, литера «.» может определять вещественную часть численной константы, либо бы частью составного оператора «..», который используется при определении диапазона значений. В этом случае может возникнуть необходимость откатится к предыдущем символу. Код метода представлен ниже:

private char GetPrevChar()

{

if (col > 0)

return buf[--col - 1];

return ' ';

}

Для нахождения конца текущей лексемы используется метод SearchCurLexem(), которому в качестве входного параметра передаётся предикат для объекта string. Переданный предикат определяет условие остановки поиска текущей лексемы. Пока условие остановки не наступило текущая литера присоединяется к строке curLexem и вызывается метод GetNextChar(), который меняет текущий элемент на следующий. К примеру, поиск конца токена операции осуществляется следующим образом:

SearchCurLexem(lex => CToken.operationMap.ContainsKey(lex + curChar));

Вместо параметра lex на каждой итерации будет подставляться значение строки curLexem. Реализация данного метода представлена ниже:

private string SearchCurLexem(Predicate<string> condition)

{

string curLexem = "";

do

{

curLexem += curChar;

curChar = GetNextChar();

}

while (condition(curLexem));

return curLexem;

}

Основное время было уделено реализации метода GetNextToken(). Это основой метод лексического анализатора, который формирует токен. В результате вызова данного метода возвращается либо токен, либо ошибка, либо null, когда достигнут конец исходного файла. Все ошибки, возникшие в ходе формирования токена, добавляются в список ошибок. Информация о них заносится в листинг программы.

Выполнение метода начинается с проверки того, что текущий элемент не является символом конца файла. Пробелы и символы переноса строки удаляются и не анализируются. Рассмотрим порядок разбора константных значений, идентификаторов, ключевых слов и операторов.

**Распознавание числовой константы.** Если текущая литера curChar является цифрой, то начинается разбор числовой константы.Будем использовать две строковые переменные wholePart и fractPart для хранения целой и дробной частей числа соответственно. Пока следующей литерой является цифра, то добавить её к строке wholePart. Далее определяем тип числовой константы. Если curChar есть символ «.», то добавим его к fractPart и считаем следующую литеру, иначе перейдём к разбору целочисленной константы. Если следующая литера вновь символ «.», то возвратимся к предыдущей литере и очистим переменную fractPart. После чего перейдём к разбору целочисленной константы. В противном случае, константа является вещественной.

**Разбор целочисленной константы.** Преобразуем строковое представление числа (wholePart) в целое число. Если во время преобразования произошла ошибка, то вызываем метод ThrowError() с ошибкой "Error in integer constant". Иначе, возвращаем токен целочисленной константы.

**Разбор вещественной константы.** Пока следующей литерой является цифра, то добавлять её к строке fractPart. Если curChar есть буква «e» (без учёта регистра), то число может быть записано в экспоненциальной форме. Добавляем curChar к fractPart и считываем следующую литеру. Если считанная литера является символами «-», «+» или цифрой, то добавить её к fractPart и вновь считать следующую литеру. Пока curChar является цифрой, то добавить её к строке fractPart.

Преобразуем строковое представление числа (wholePart + fractPart) в число с плавающей запятой. Если во время преобразования произошла ошибка, то в строковом представлении числа есть недопустимая литера. Вызываем метод ThrowError() с ошибкой «Illegal character». Если значение получившейся константы превышает максимальное допустимое значение типа Real, то возвращаем ошибку «Error in real constant». Иначе, возвращаем токен вещественной константы.

**Распознавание идентификатора или ключевого слова.** Если текущая литера curChar является буквой или символом «\_», то пока следующая литера есть буква или символ «\_» - добавляем литеру к строке name и считываем новую литеру. Если name является ключом словаря operationMap, то вернуть токен операции. Иначе вернуть токен идентификатора.

**Распознавание оператора.** Если текущая литера curChar является ключом в словаре operationMap, то пока добавление следующей литеры приводит к образованию составного оператора – добавляем литеру к строке oper и считываем новую литеру. После чего вернуть OperationToken с типом операции operationMap[oper].

**Распознавание строковой константы.** Если текущая литера curChar - кавычка, то пока следующая литера не является закрывающей кавычкой или не будет достигнут конец строки – добавляем текущую литеру к строке strConst и считываем новую литеру. Если закрывающая кавычка не встретилась или длина строковой константы больше 255, то вызвать метод ThrowError() и выдать ошибку «String constant exceeds line». Иначе вернуть токен строковой константы.

**Пропуск комментариев.** Если текущая литера curChar – открывающая фигурная скобка, то пока следующая литера не является закрывающей фигурной скобкой или не будет достигнут конец файла – считываем новую литеру. Если закрывающая фигурная скобка не встретилась, то вызвать метод ThrowError() и выдать ошибку "Unexpected end of file". Иначе получить следующую литеру и вернуть GetNextToken().

Если текущая литера не соответствует ни одному из описанных выше случаев, то распознать лексему не удалось. Работа метода GetNextToken() заканчивается выдачей ошибки «Illegal character».

# Тестирование

В этом разделе протестирована работа лексического анализатора на примере кода, содержащего различные конструкции языка, а также несколько лексических ошибкок

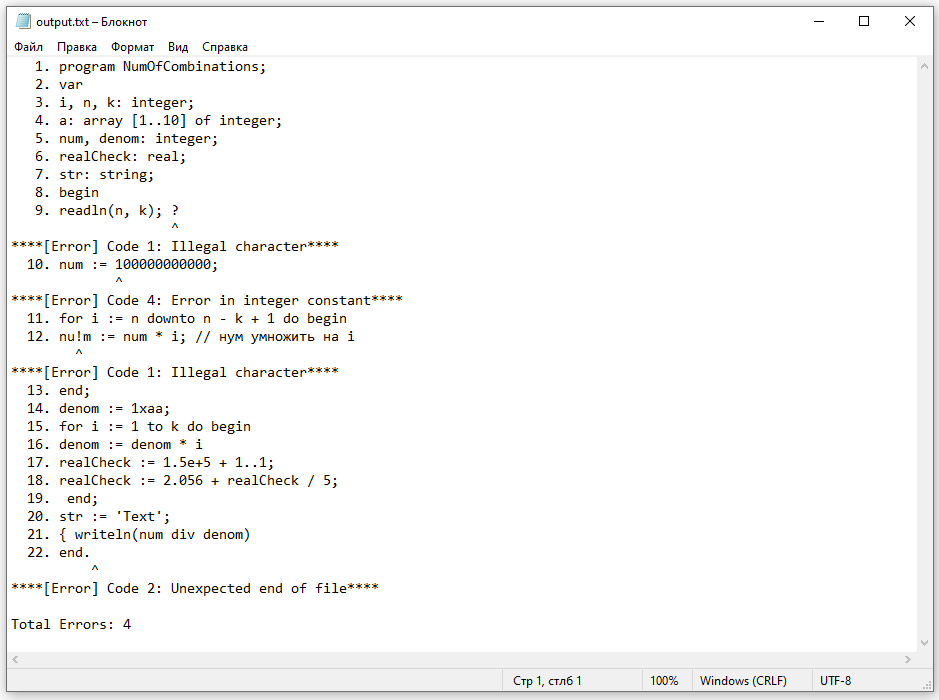
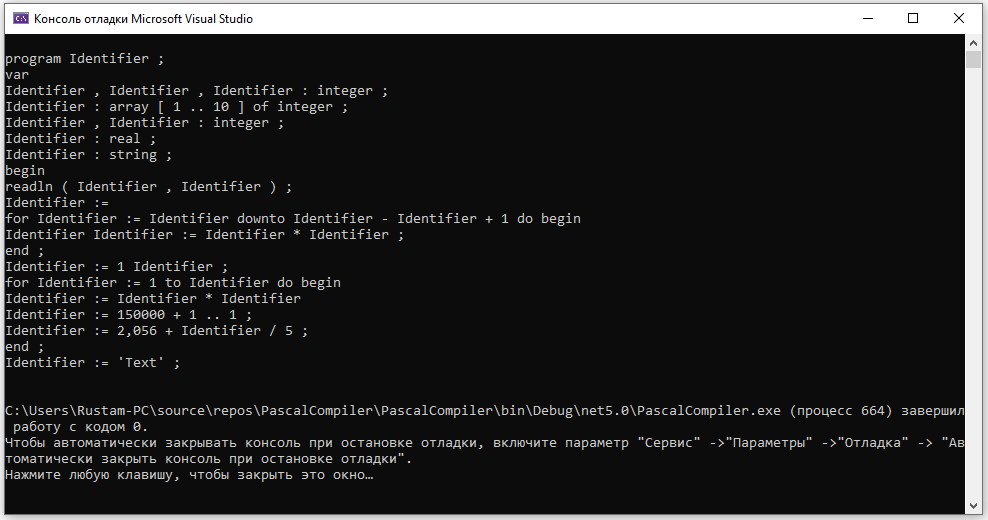


Рисунок 2 – Результат работы лексического анализатора и листинг ошибок